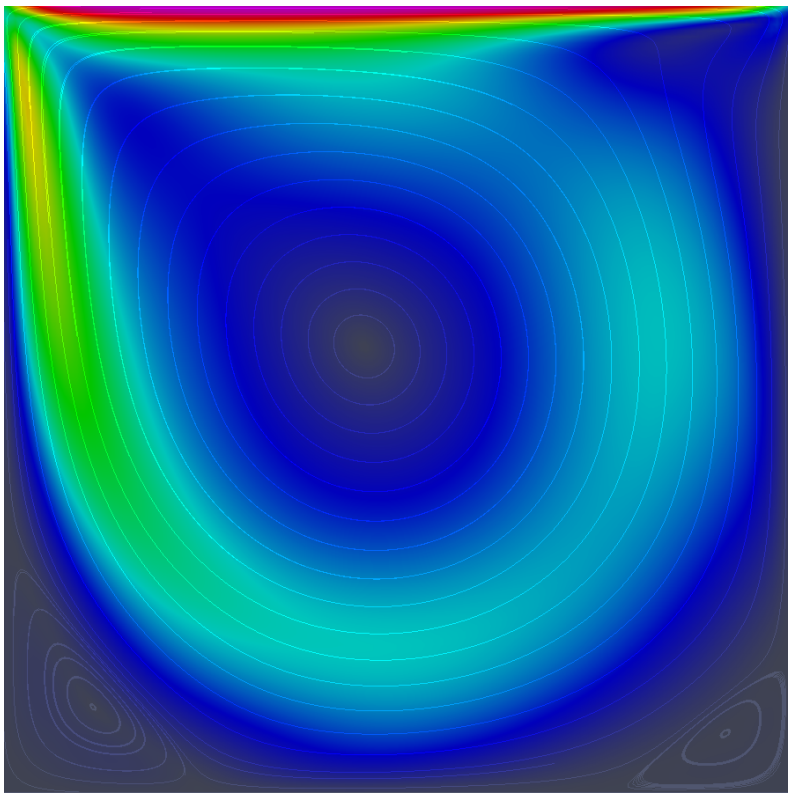
CHALMERS



An Incompressible Navier-Stokes Equations Solver on the GPU Using CUDA

Master of Science Thesis in Complex Adaptive Systems

NIKLAS KARLSSON

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, August 2013

An Incompressible Navier-Stokes Equations Solver on the GPU Using CUDA
NIKLAS KARLSSON

Examiner: Ulf Assarsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Master's Thesis 2013:08
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31 772 1000

Cover:
GPU accelerated simulation of the 2D lid-driven cavity problem.

Department of Computer Science and Engineering
Göteborg, Sweden, August 2013

MASTER OF SCIENCE THESIS IN COMPLEX ADAPTIVE SYSTEMS

An Incompressible Navier-Stokes Equations Solver on the GPU Using CUDA

NIKLAS KARLSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden, 2013

An Incompressible Navier-Stokes Equations Solver on the GPU Using CUDA
Master's Thesis in Complex Adaptive Systems
NIKLAS KARLSSON
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Graphics Processing Units (GPUs) have emerged as highly capable computational accelerators for scientific and engineering applications. Many reports claim orders of magnitude of speedup compared to traditional Central Processing Units (CPUs), and the interest for GPU computation is high in the computational world. In this thesis, the capability of using GPUs to accelerate the full computational chain of a 3D incompressible Navier-Stokes solver, including solvers and preconditioners for sparse linear systems as well as assembly routines for a finite volume discretization, has been evaluated. The CG, GMRES and BiCGStab iterative solvers have been implemented on the CUDA GPGPU platform and evaluated together with the Jacobi, and Least Square Polynomial preconditioners. A double precision Navier-Stokes solver has been implemented using CUDA, adopting a collocated cartesian grid, SIMPLEC pressure-velocity coupling scheme, and implicit time discretization.

The CUDA GPU implementations of the iterative solvers and preconditioners and the Navier-Stokes solver were validated and evaluated against serial and parallel CPU implementations. For the iterative solvers, speedups of between six and thirteen were achieved against the MKL CPU library, and the implemented methods beats existing open source GPU implementations of equivalent methods. For the full Navier-Stokes solver, speedups of up to a factor twelve were achieved compared to an equivalent commercial CPU code when equivalent iterative solvers were used. A speedup of a factor two was achieved when a commercial Algebraic MultiGrid method was used to solve the pressure Poisson equation in the commercial CPU implementation.

The bottleneck of the resulting implementation was found to be the solution of the pressure Poisson equation. It accounted for a significant part of the total execution time for large problems. The implemented assembly routines on the GPU were highly efficient. The combined execution time for these routines were negligible compared to the total execution time.

The GPU has been assessed as a highly capable accelerator for the implemented methods. About an order of magnitude of speedups have been achieved for algorithms which can efficiently be implemented on the GPU.

Keywords: GPU, GPGPU, CUDA, Iterative Solver, Preconditioner, Navier-Stokes Equations, Fluid Solver, CFD, Finite Volume Methods

Preface

This thesis is partial fulfillment of the requirement for the MSc degree from Chalmers University of Technology. The work was carried out during the period February 2013 to August 2013 at the Fraunhofer-Chalmers Centre (FCC). Associate Professor Ulf Assarsson at the Department of Computer Science and Engineering, Division Graphics was examiner and Dr. Andreas Mark at FCC was supervisor.

Acknowledgements

This work was done in the project "The Virtual Paint Factory" with partners Volvo Car Corporation, Scania CV, AB Volvo, Swerea IVF and Fraunhofer-Chalmers Centre. The project is funded in part by VINNOVA through the FFI Sustainable Production Technology program, and in part by the Sustainable Production Initiative and the Production Area of Advance at Chalmers.

I would like to direct a special thanks to my supervisor Dr. Andreas Mark for his expertise, optimism, and encouraging help and support. I would like to thank Associate Professor Fredrik Edelvik and Dr. Johan Carlsson for believing in me with this master thesis work and other challenging assignments, for your professional guidance and for putting faith in mine and other young students, and researchers abilities. I would also like to thank my examiner Associate Professor Ulf Assarsson for taking on this project and providing valuable input. I owe a debt of gratitude to Lic. Björn Andersson and Lic. Anders Ålund for fruitful discussions, valuable input and computer support. Last but not least I would like to thank all my other colleagues for keeping their doors open and making me feel at home at FCC.

Niklas Karlsson, Gothenburg 28/8/13

Contents

Chapter 1

Introduction

The Navier-Stokes equations are used to model fluid flow in numerous scientific and engineering applications. It is used in climate research and meteorology to model weather systems; in medicine to model blood flow; and in engineering to aid in the design of cars boats, and airplanes. Solving real world fluid flow problems are often computationally heavy, both in terms of time and required memory. Simulations can easily take days, weeks or months to finish. Speeding up such simulations with an order of magnitude could be ground breaking for many applications, allowing designers to work interactively with applications and use models more extensive and accurate.

With the arrival of modern graphics card processors, the computational power of normal desktop computers have increased by an order of magnitude. For many years, this extra computational power was utilized almost exclusively for visualization of graphics applications. Today, things are starting to get different, cutting edge engineering and scientific applications are using the extra processing power coming from the graphics processor to achieve orders of magnitude speedup. The worlds fastest supercomputers are built with Graphics Processing Units (GPUs) for increased computational power and reduced power consumption [1]. However, the speedup does not come automatically. GPUs consist of many cores allowing for massively parallel computation, but also requires massively parallel applications. Specific programming platforms and models, such as CUDA and OpenCL, have emerged to allow for general purpose GPU programming.

Navier-Stokes equations can allow for massive parallelization if discretized properly. Using a Finite Volume method on a structured grid the discretization of the equations allow for locality in assembling properties in each computational node, suitable for parallel computations. To solve the discretized equations, one typically have to solve a large sparse linear system, $Ax = b$, where the majority of the elements in the coefficient matrix A are zero. An efficient way of solving such a system is to use an iterative method. Rather than directly computing the inverse of the matrix, as would be the case for a direct method, the iterative methods gradually converges towards the true solution stopping once the residual is small enough. The main operation of such methods is the sparse matrix-vector multiplication operation, which can be massively parallelized.

1.1 Scope of this thesis

The aim of this thesis work is to design, implement, evaluate and optimize a complete Navier-Stokes equations solver on the GPU. This involves finding, efficiently implement-, and evaluate suitable methods for solving large sparse linear systems on the GPU; along with suitable discretization and assembling techniques of the Navier-Stokes equations on the GPU.

This work is limited to using the CUDA, Compute Unified Device Architecture, GPU programming platform. Other languages such as OpenCL have not been explicitly evaluated. The choice of GPU used for computations, was partly based on the GPU available to the author. It was neither the best nor the worst on the market, and the performance when using other GPUs was not evaluated but could only be estimated. Furthermore, as the main aim with the work is to assess the capability of using GPUs to accelerate a Navier-Stokes solver, the ability to handle arbitrary computational domains has not been of high focus.

Motivating this research is the potential in improving the performance of the state of the art general purpose fluid flow solver IBOFlow [2, 3, 4, 5],s developed at Fraunhofer-Chalmers Research Centre for Industrial Mathematics (FCC) [6].

1.2 Approach and accomplished work

This thesis project has been posed with the open question of evaluating the GPUs potential in speeding up a finite volume discretized, implicit time, Navier-Stokes equations solver. The open question gives a high degree of freedom but also a demand for responsibility and ability for independent work. The aim of this thesis when starting it was that the final product, if successful, would be a solid foundation of a complete general purpose GPU implementation of the Navier-Stokes equations. This includes methods, techniques and evaluation of limitations as well as a significant code base. The solver would then be incorporated into IBOFlow. Therefore inspiration from the high level code structure of IBOFlow was used in the development of the code to ease an eventual incorporation process.

Attaning and gathering in-depth knowledge and expertise of GPU computing and CUDA has been a key component and essential to carry through this project efficiently, and thereby a non-negligible part of the work.

Iterative methods were evaluated on the GPU since a large part of the potential speedup of the Navier-Stokes equations lies in performing these routines efficiently. The sparse iterative solvers GMRES, CG, and BiCGStab; together with the least square polynomials, and Jacobi preconditioners were implemented as a part of this thesis work. Building blocks for these methods were elementary linear algebra routines, such as vector-vector multiplication and sparse matrix-vector products (SpMV), from Nvidia's cuSPARSE and cuBLAS libraries [7]. Sparse matrix formats and different SpMV implementations were evaluated, but there was eventually nothing motivating an implementation of these methods specifically for this project.

The full set of discretization and assembly routines for solving the Navier-Stokes equations were implemented on the GPU as a part of this thesis.

The choice of implemented iterative solvers and preconditioners as well as the choice of discretization and coupling schemes for the Navier-Stokes equations were based on literature studies.

1.3 Organization of the report

The structure of this report is as follows: In Chapter 2, the Graphics Processing Unit (GPU) alongside with the GPU programming platform this work is based upon, CUDA, are introduced and described in detail. First, GPU history and differences to the Central Processing Unit (CPU) is described. It is followed by the GPU architecture, programming model, and optimization considerations. A specific CUDA programming and optimization example is given in the end of this chapter.

In Chapter 3, previous work within the fields of sparse linear algebra, sparse iterative methods, and Navier-Stokes equations solvers on the GPU are referred to.

In Chapter 4, implementations of the Sparse-Matrix Vector operation are described and results presented.

In Chapter 5, sparse iterative solvers and preconditioners are described. Algorithms of the implemented iterative methods and preconditioners is given, together with an idea of the underlying theory and application, and implementation considerations. The results of the iterative methods and preconditioners are also presented in Chapter 5.

The approach, method and discretization schemes for the implemented incompressible Navier-Stokes equations solver are described in Chapter 6. CUDA considerations and optimizations used for these routines are also discussed. The chapter ends with presenting the achieved results.

In Chapter 7, conclusion, complementary discussion, and outlook are given.

Chapter 2

The Graphics Processing Unit

The Graphics Processing Unit (GPU) is, like the Central Processing Unit (CPU), designed with the sole purpose of performing calculations. Unlike the CPU which is designed at performing general computations, the GPU is specialized at processing graphics data visible to the user through the screen. Graphics processing involve processing huge amounts of data in parallel. The GPU has been adopted to suit this purpose, resulting in a design with many computing cores which can perform computations in parallel, together with a high memory bandwidth. As it seems, such devices can be utilized to accelerate also other computations than graphics processing, such as engineering calculations. Accelerating engineering computations is what brings the GPU into the scope of this thesis. Programming GPUs effectively require some basic knowledge of its underlying architecture and functionality. This chapter will start with describing the GPU in general, followed by the specific GPU programming platform, CUDA, used in this thesis.

2.1 General Purpose-GPU: Past, present and future

The GPU was originally designed for accelerating graphics rendering, which it has been optimized for. Graphics rendering is in nature highly parallel, involving local operations on large sets of pixels and vertices. The first programming interfaces for GPUs were via graphics-, also known as shader-, languages in OpenGL and DirectX. These graphics programming languages were where the idea of General Purpose GPU (GPGPU) computing was first explored and where its potential was discovered. For the first graphics processors, OpenGL and DirectX only supported a fixed modifiable, but not programmable, API pipeline. The first steps towards a programmable pipeline, were taken in 2001 at the time of DirectX8 [8]. The first GPGPU programming was carried out in a very limited fashion. The possible operations in early GPU architectures were restricted to those required for graphics operations. As an example, locations of memory writes were restricted and floating point data was not supported [9].

Long before the arrival of modern GPGPU languages such as CUDA, GPGPU applications were developed using the shader languages. Confirming this, the website

"GPGPU.org" [10] were founded as early as 2002. Predecessors to todays GPU programming platforms were the high level GPU languages BrookGPU [11] and SH [12], launched in 2004. BrooksGPU is an extension of the C programming language developed at Stanford University, and SH is an extension to C++ developed at Waterloo University, both built on top of shader languages.

Many attractive features for GPGPU applications of todays GPUs originates from the need of a similar kind of feature or improvement required from graphics developers. An example being the high memory bandwidth, which was required for high resolution displays to be able to update the frames rapidly. Memory bound GPGPU applications are common. In these applications, memory bandwidth more than the peak Floating point Operations Per Second (FLOPS) rate is what gives the GPU an advantage over the CPU.

Modern graphics programming took a leap forward when Nvidia released their G80-series in the year shift 2006/2007, built on what is called the Tesla architecture. It opened up the world of GPGPU computing to a wider public. A new combined hardware and software architecture called Compute Unified Device Architecture (CUDA) was introduced, simplifying the programmability and GPGPU possibility of the GPUs through both hardware changes and software support.

CUDA introduced several new features: A unified shader pipeline, where each Arithmetic Logic Unit (ALU) could be programmed freely. The arithmetic units had built in support for single-precision floating point arithmetic complying with IEEE standards. The set of callable instructions were more adopted to general GPU computations. The programmer was given read and write access to arbitrary memory locations on the GPU. A shared memory acting as a manually controlled cache was introduced. Also parallel programming primitives such as threads, atomic operations, and synchronization were made available. Several months after the initial release of CUDA, an extension to the C programming language with a callable API to the GPU functionality was made available to developers.

There exist alternatives to CUDA. Most notably OpenCL, which is open source and supervised by the Khronos group, and Microsofts DirectCompute and C++AMP. The GPU has found its way into todays supercomputers, which are increasingly using GPUs for speeding up applications and reducing power consumption. The TITAN supercomputer of the Oak Ridge National Laboratory [1] is one of the worlds fastest computers, and iss built with both GPUs and CPUs. The increased use and threat from GPU manufacturers in the super computer industry has made Intel come up with a competing device, the Xeon Phi accelerator. The interest of major developers, such as: Intel, Nvidia, and AMD, in the many core architecture super computing industry, promises good future development in the area. Both GPUs and the GPGPU phenomena are relatively new, and significant future improvements and discoveries can be expected. On the hardware side, researchers will find ways to do calculations more efficient and open up new programming possibilities. On the application side, speedups of yet more applications can be expected as researchers in different fields and with increased GPU proficiency accelerates their applications with GPUs.

2.2 A closer look: the GPU versus the CPU

To start with, the basic functionality of a CPU will be briefly explained. The CPU is a hardware unit which is built for performing arithmetic, logic and input/output operations of a computer. It performs the instructions a programmer has written as a program. The CPU consists of execution units, control logic, and caching and is connected to a larger Random Access Memory (RAM), of size in the orders of GBs. The clock frequency of a CPU is a measure of how many operations it can perform per second, but it is not the only factor limiting performance. In many applications, a more important measure is the memory bandwidth. In order to perform an operation, both the operation and the data to perform the operation on must be loaded onto the CPU. Memory bandwidth describes how fast the memory transfer between the CPU and the RAM memory is. To improve the performance of often accessed memory locations, relatively large amounts of data is stored in the CPU's caches. The reason caches are used, is that memory accesses from RAM takes many clock cycles while access from caches takes only a few. The caches of a CPU are divided into a hierarchy with faster but smaller caches closer to the arithmetic and logic processing units and larger but slower further away. An important concept for achieving high performance of the CPU is to avoid cache misses. These occurs when a memory location required for computation is not found in the caches, but rather need to be accessed from the RAM memory directly.

A natural question to ask when reports are claiming several orders of magnitude of speedup for GPUs compared to CPUs is: what makes the GPU more capable of computations than the CPU, and for what applications does this apply? The general answer is that they have different design targets and thus the transistors are distributed differently between different tasks. The aim of the GPU is to optimize parallel processing power, without having to devote large amounts of transistors to control logic and caching. The CPU on the other hand is general purpose and optimized for serial code, and performs well on most tasks. To be general purpose is a tradeoff to peak performance, but it results in a more flexibly programmed processor. More specifically, the CPU has more transistors devoted to control logic and caching, while the GPU has more devoted to computing power involving only rather basic control logic. The CPU will perform better on serial programs while the GPU is better suited for highly parallel programs.

The GPU demands more of the developer to be utilized optimally. It also requires different programming ideas and platforms than the CPU due to its parallel architecture and execution. For example, the CPU has extensive control logic and caching which effectively can remove memory access latencies. The GPU however can also reduce memory access latencies, but to do so requires thousands of parallel threads to be launched simultaneously, performing computations on some threads meanwhile fetching memory for other threads. A major architectural and practical concept is that the GPU is built with a Single Instruction Multiple Data (SIMD) philosophy, being able to coalesce memory access and instruction fetches if programmed correctly.

To continue the comparison, let us compare some technical data of a high end CPU, a high end GPU, and the GPU used for computations in this thesis. The high end CPU

is the Intel Xeon E5-2650 (also used for CPU computations in this thesis work), the high end GPU is the Nvidia Geforce Titan. Both are available at a retail price of about 1000 USD. The GPU used in this thesis is a Nvidia Geforce 660 Ti. It is available at a retail price of around 300 USD. The technical data is compared in Table 2.1. The significant differences are the memory bandwidth, peak FLOP performance, and powerr efficiency, which are higher on the GPU. Not visible in the table are cache sizes, control logic etc. which are more sophisticated on the CPU.

Table 2.1: Comparison of an Intel CPU used in this work and two different Nvidia GPUs. The Geforce 660 Ti is the GPU used for calculations in this thesis work.

	Geforce Titan GPU	Xeon E5-2650 CPU	Geforce 660 Ti
Cores	2688	8	1536
Clock Speed [MHz]	836.5	2000	915
Memory bandwidth [GB/s]	288.4	51.2	144.2
TDP [Watt]	250	95	150
Memory (max) [GB]	6	750	3
No of ransistors [10^9]	7.08	2.27	3.54
Retail price	$1000	$1100	$300

The GPU approach of using many cores with low clock rate rather than a few with high, is beneficial from performance aspects. There is a limitation of how high clock rate for CPUs that can be produced. Further the power consumption increases super-linear with clock frequency, in addition causing difficulties with cooling and lower efficiency. Estimates have been given roughly approximating that GPUs are 10 times more energy efficient than CPUs [13].

Not excluding either the GPU nor the CPU it is an appealing option to use hetereogenuous hardware architectures, which means combining the GPU and CPU. The CPU and GPU can perform computations at the same time and are good at different tasks, complementing each other. The supercomputers using GPUs are often build with this strategy in mind. The drawback with GPU-CPU programs is that the memory transfer between the GPU and CPU is slow.

2.3 The General Purpose GPU: An overview

This section gives an overview of the GPU architecture from a GPGPU perspective. More details and specifics of interest to developers can be found in Section 2.4, where CUDA is described.

The GPU consist of streaming multiprocessors (SMs) which are connected to the Dynamic Random Access Memory (DRAM), the global memory or simple GPU memory, of the GPU. Note that the DRAM of the CPU and GPU are different. A stream in

computer science is a set requiring similar computations. The SMs in turn consists of many single-instruction-multiple-data units (SIMD), also referred to as stream processors. In the stream processors one instruction is executed at a time but over several data sets independently. This is a major benefit of the SIMD architecture, and implies that many instructions and memory operations can be combined and coalesced. Many GPUs traditionally perform operations in a vectorized fashion. This has led to that some operations consisting of several add and multiply operations are in the GPU treated as one operation. An example is the addition of two RGB colors. The GPU is commonly connected to the CPU via 8-16 PCIe3 buses. This can be a major performance bottleneck needed to take into consideration when transferring data.

2.4 CUDA

The thesis work presented in this report is based on the CUDA programming platform, which will be explained in detail in this section.

CUDA is a programming platform developed by Nvidia to facilitate general purpose computations on Nvidia GPUs. The CUDA programming model is based on Single Program Multiple Data, SPMD, operations. This means that the same operations are carried out by several threads but acting on different data. In a data independent for-loop, the CUDA structure would be that each thread carries out the body of one or a few loop iterations. The CUDA programming interface is essentially a C/C++ based programming interface for communication with GPUs, extended with GPU programming functionality similar to C/C++.

The first CUDA architecture was the Tesla architecture. Its main features have already been briefly explained in the GPGPU history chapter. Double precision floating point operations support was introduced on later Tesla GPUs. After Tesla, the Fermi architecture was launched in 2010. It introduced among others: more peak double precision Floating Point Operations Per Second (FLOPS), support for double precision floating point numbers complying to IEEE standards, caching, debug support, and improvements of many existing features. In 2012, the Kepler architecture was introduced. It is the most recent architecture at the time of the writing of this report. New features in the Kepler architecture were: Dynamic Parallelism - allowing kernels to call other kernels, thus supporting recursive calls; Hyper-Q - allowing up to 32 CPU threads to launch work on the same GPU, facilitating MPI applications; and GPUDirect - allowing other GPUs and other RDMA equipped devices to directly access the GPUs memory without passing via the host device. Also the power consumption was significantly reduced.

On Nvidia's GTC 2013 conference their CEO gave some insight into future GPU developments in his keynote speech. The coming architecture will be called Maxwell, and will introduce a feature allowing GPUs and CPUs to read from each others address spaces. The Maxwell is said to be expected in 2014. The successor of Maxwell is said to be Volta. It will introducing what appears like a significant improvement for many GPGPU applications, namely stacked DRAM of the GPU. This means that the DRAM will be stacked on top of the computing chip on top of each other and communicate to

each other via a silicon substrate. According to Nvidia this will lead to 1 TB/s of DRAM memory bandwidth compared to todays 200-300 GB/s. Also for every layer stacked memory on top of each other the memory capacity will increase by that many times. Nvidia have also said that they are working on enabling more programming languages via an open source GNU compiler back-end. Other software related features to be expected in future CUDA versions are: just in time linking and compiling, ARM-device support, supporting c++11, a library of sparse linear algebra solvers, and single GPU debugging. Even further in the future we can expect optimized memory locality and computation as well as task parallelism. Furthermore Nvidia will introduce CUDA into their Tegra series mobile chip. The first such GPU will be called Logan and is said to arrive in the year shift 2013/2014.

2.4.1 Execution hierarchy

The parallelism in CUDA follows a hierarchy. Threads are grouped together in blocks and blocks are contained in a grid, consisting of all computational parts of the kernel. A block is assigned to an SM, and resides on the same SM until it has finished its computations. Threads in the same block run concurrently. They can communicate via block synchronization and shares a programable memory space, similar to the CPU's L1 cache, called shared memory. Once all threads in the block have finished computations the block has finished. An SM can have several blocks assigned to it at the same time and schedules work between them as it finds suitable. Some blocks might run in parallel but not necessarily all, depending on the resources available. It is required that all blocks can be executed independently and in any order. The kernel has completed its computations once all blocks have finished their computations.

Thread blocks consist of several sets of 32 threads called warps, which are executed by stream processors. A warp is ideally computed in parallel as a Single Instruction Multiple Data, SIMD, unit. In SIMD, the same instruction is performed by each thread within the warp but the data varies between threads. This design makes it possible for coalescing of several operations for all threads in a warp, such as instruction fetching and memory fetching. All threads in a warp are executed in a lock-step fashion, which means that only one instruction is performed at a time. If all threads are to perform the same instruction then this execution is in parallel. However, if some threads in the warp are to perform different instructions, this gives rise to what is called thread divergence. Thread divergence can be caused if threads within a warp follow different branches of a conditional statement. In the case of thread divergence, the execution will be serialized to as many instructions as there are diverging branches. Threads within the branch that do not perform the executed instruction will remain idle. To allow for thread divergence in CUDA makes the warps execute in what is called Single Instruction Multiple Threads (SIMT). It differs from the previously mentioned SIMD by allowing threads to take different execution paths.

Dividing work into blocks and threads is not explicitly required of the programmer. The programmer rather writes the code as SPMD, meaning that the same piece of code can be executed by any of the threads, using different data, with the possibility to

extinguish different threads and blocks. More low level control, such as which threads get scheduled to which SM and in which order the threads get executed, is handled by CUDA.

2.4.2 Memory hierarchy

The memory of the GPU follows a hierarchy. Private to individual threads are registers. Shared within a thread block is an on-chip L1 cache. The L1 cache consists of a programmable "shared memory" and a general purpose cache, the latter is used to speed up random access operations. Shared memory is completely software managed. The partition of L1 cache used for shared memory and general purpose memory respectively, can be determined by the programmer. The size of the total L1 cache is 64KB on Kepler and Fermi GPUs. The shared memory part of the L1 cache opens up intercommunication between threads in a block.

Further there exist read only data caches: texture cache and constant memory cache. Texture and constant memory are visible to all threads of a kernel. The beneficial access pattern for constant memory is that all threads within a warp should access the same memory location. For textures, there should be spatial locality of data, no uniform access is required. The texture memory is hardware managed. Shared to all blocks on an SM is an L2 cache, which is of size 1536KB on Kepler. The L2 cache is a general purpose cache and speeds up code where the access pattern is not known beforehand. Shared within the whole GPU is the DRAM, also called global memory. The size of the global memory is up to 6GB with a bandwidth of up to 288GB/s at the time of writing this report. The global memory of the GPU has higher bandwidth but also slightly higher memory latencies than its CPU equivalent.

Warp operations play an important role also for memory accesses. Warps can coalesce memory fetches from global memory, if the memory locations accessed are contiguous. More specifically the global memory is organized into segments of 128 bytes, all threads within a warp which access memory from the same segment will be coalesced. Memory access is fully coalesced if all threads in a warp accesses the same segment, and uncoalesced if all threads in a warp access different segments. The number of memory transactions needed for a warp, is the number of different segments which that warp fetches memory from. Also memory fetches from caches can be coalesced. Generally memory closer to the threads have faster read and write access speeds. For instance, shared memory is about hundred times faster than global memory. Access latencies from global memory is hundreds of clock cycles. Coalesced memory access is required for reaching peak memory bandwidth.

When programming the shared memory one has to be aware of what is called a memory bank conflict. Bank conflicts arise in shared memory when two threads try to access the same memory bank. The cost of a bank conflict is the maximum number of simultaneous accesses to a single bank. This should be avoided since such requests will be executed sequentially after each other. However, if all threads in a warp access an identical address, this will not lead to a slowdown [14].

The architectural benefit of dividing threads into warps, comes from that memory

and instruction access can be coalesced. The cost for doing such an operation can thus be amortized over all threads in the warp.

2.4.3 Hardware limitations of the streaming multiprocessor

The Streaming Multiprocessor, SM, is subject to hardware restrictions, which limits the size of threads and memories residing on it. Some limits are hardware specific and the following numbers applies to Kepler GPUs, even though many numbers are the same for previous architectures. The maximum number of threads per block is limited to 1024, and the maximum number of blocks simultaneously scheduled to an SM is limited to 16, while the maximum number of threads of an SM is limited to 2048. Registers are also distributed within an SM, and is limited to 65536 per SM. The maximum number of registers per thread is 255. The shared memory is also of fixed size within the SM, being 64KB, as already mentioned.

When programming CUDA one needs to be aware of these restrictions as they might limit performance. None of these hardware limits may be violated. This leads to that if the maximum number of register for an SM has been reached, no more blocks can be scheduled to that SM, and the throughput is possibly reduced. If any property would be violated, then the number of blocks of the SM is reduced by an integer number. That is, the limiting adjusting building block is the blocks scheduled to it. Imagine the following scenario. The number of registers per thread is increased by one, and that this increase causes one block less to be able to be scheduled on an SM due to that the maximum number of registers is reached. Assume further that the number of threads per block is 1024. Then by reducing the registers by one, two blocks of 1024 threads can be scheduled to the SM rather than one. This change in a single register can thereby theoretically increase the performance by almost a factor two. An analogous limit applies for shared memory. To control the occurrence of this issue, one can set a limit on the maximum number of registers per thread and shared memory per block by specifying it to the Nvidia CUDA compiler, NVCC.

2.4.4 A scalable programming model

In computer science parallelism can conceptually be split up into different levels describing how often the parts being executed in parallel communicate with each other. Coarse grained parallelism refers to that a problem can be divided into several independent sub problems. Fine grained parallelism refers to performing even smaller subproblems in parallel. Performing two matrix vector products of two independent vectors with the same matrix could be considered coarse grained parallelism while computing the accumulated sum of a vector would require fine grained parallelism. CUDA supports such different levels of parallelism. Kernels can be computed in parallel, solving different tasks (task parallelism), while many threads can compute sub problems of a task in parallel. Threads within a block can communicate with each other via shared memory and thread barrier synchronization. Different blocks can communicate once the control is given to the CPU, launching the kernel again with different data if necessary. Threads, blocks and kernels

comprises a scalable programming model, where the programmer does not need to know the exact number of available threads or blocks when programming. Many architecture specific properties, such as the limitations of an SM, can be queried at runtime. The scalability is an important concept as it generalizes the task of programming many core devices, and allows for code written today to be efficient on future devices, including those with orders of magnitude more computational cores.

2.4.5 The CUDA programming language

CUDA device code is based on C/C++, but it does not support all features from recent standards [15]. The code flow consists of what is called host code, usually ran on the CPU, and device code, called kernels, ran on the GPU. From now on it will be assumed that the host is a CPU and that the device a GPU. The CPU part of the program takes care of program control: making calls to the GPU for allocating memory and transferring memory between GPU and CPU, invoking calls to GPU kernels, and synchronizing the operations performed on the GPU. It is a single sided communication where the host can send commands to the device but not vice versa. The host and the device have different memory spaces. They cannot directly access each others memory. Kernels were initially solely launched from the host code, but with the introduction of dynamic parallelism on the latest Kepler architecture, kernels can be called from within kernels. In order to make a kernel call, one has to specify certain multithreading parameters. Most of the GPU and CPU computations can be done in parallel.

The host code is compiled with a standard C/C++ compiler, whereas all files containing device code and kernel invocations are compiled with a CUDA compiler. The compilation of CUDA code can be separated in two steps. Either one can compile directly to binary code, or one can compile to some intermediate PTX assembly format. Binary code is architecture specific, the PTX format is not. PTX code is compiled at runtime to device code by the device driver. The PTX format enables code to be compiled for architectures not yet built by the time of the writing of the code, and also benefit from new features becoming available at later times. CUDA wrappers exist for other languages/platforms than C/C++ so that CUDA can be written in for example Python, Fortran, Java and Matlab.

Atomic operations is an important concept for thread safety and parallel computing. During the duration of an atomic operation the state of the data being read or written to, cannot be changed by any other thread. Several atomic operations are supported on CUDA devices.

CUDA comes with several useful optimized software libraries provided by Nvidia. Of interest to this thesis work are the libraries for Basic Linear Algebra Subroutines (BLAS): cuBLAS, and cuSPARSE. Other libraries of interest to scientific computing are cuRAND, cuFFT, and the library THRUST providing parallel implementations of basic routines.

2.4.6 Programming features and performance considerations

For efficient and high performance execution, the CUDA code typically has to be carefully developed and optimized, which often is a non-trivial task. With the described GPU and CUDA architecture as a foundation, this section will explain some general optimization considerations used in this thesis work.

In CUDA, the programmer must choose several parameters for execution. The most notable example is that whenever a kernel is invoked, the programmer must specify the number of threads per block and the total number of blocks it should be launched with. The choice of these parameters has impact on the performance of the kernel, and while there are no deterministic rules of what will be most efficient there exist some general guidelines. To maximize coalescing of warp operations and to minimize thread divergence, the number of threads per block should be a multiple of the warp size. To cover up for memory latencies, many threads should be scheduled to each SM, which implies that a large number of threads per block should be used. To set the number of threads per block to 128 or 256 is usually a decent choice on todays architectures. Have in mind that many GPU specific details are available to the application at runtime.

Many applications are memory bound. This makes it important to optimize the use of different memory spaces when optimizing code. Generally one should make use of all low level memory spaces available, as locality is an important concept for high performance. Registers can be used to maximize data reuse, leading to lower reads and writes from global memory. Shared memory can be effective when the access patterns of threads are known beforehand, and when threads within a block make use of the same elements but at different instructions in the code (to avoid bank conflicts). Constant memory is cached and is ideal for use with program constants, but can also be used for arrays. Texture memory is optimized for spatial locality in 2D and can therefore be used if coalesced global memory access cannot be achieved. However the L2 cache on Kepler architectures supposedly manages such memory reuse effectively as well. Recomputing data rather than caching it or fetching it from global memory, should also be considered when possible as a means to reduce the number of global memory accesses.

Perhaps the most important aspect to optimize memory bandwidth, is to program for highly coalesced memory accesses for threads within a warp. As already discussed, this is possible when contiguous threads (threads within a warp) access contiguous memory spaces. An example is that one should use a struct of arrays rather than an array of structs, and arrange data structures such that accesses can be coalesced. Consider the case of having a struct where an instance of a struct stores all cell properties for one cell, u, v, w. Now when two consecutive threads perform the same instruction, say $x = u[tID]$, then the first thread will access a specific address to get its velocity, while the other thread will access a memory address being *sizeof(struct)* elements away, not being consecutive elements. This leads to that the maximum number of coalesced memory accesses by a warp is $32/sizeof(struct)$. On the other hand, if we store all velocities in a separate array. Then the first thread will access $u[tID]$ and the other $u[tID + 1]$, where $u[tID]$ and $u[tID + 1]$ are two consecutive memory addresses. This leads to that each warp is able to perform 32 coalesced accesses, being the optimal value. Aligned access

patterns are beneficial also for accessing the cache memories previously mentioned.

A technique often used together with shared memory is tiling. Threads that make use of the same elements are arranged in a block, each thread loading some memory into shared memory and then all threads make use of the whole shared memory. Now each property only need to be read once and written once for the whole block, rather than once for each thread within the block using the property. This reduces the number of read and write operations to global memory to a maximum of two per property for each block. Tiling and shared memory can also help coalescing memory access in non-structured data structures via reordering the data through it.

As already discussed, warp divergence should be avoided. However, if the branch granularity is a multiple of the warp size, this is not an issue.

Another efficient technique is loop unrolling, where the operations of several loop iterations are all performed during one iteration. This decreases loop overheads and can sometimes benefit from data reuse. Loop unrolling might require larger register usage, but can decrease the number of global memory retrievals. This technique can be efficient if there is an extra overhead for new threads, which can be removed by merging several of them together.

An issue with CPU-GPU heterogeneous programs is that memory copy between the CPU and GPU is a serious bottleneck. It is limited by a PCIe 3.0 connection. Such programs are common in large programs where only computationally intensive parts are ported to the GPU. In these cases, large portions of memory need to be copied between the CPU and GPU whenever a GPU kernel is invoked. This limits the potential benefit in using a GPU for the computation. All data required for the computation by the GPU need to be in the GPU memory, and solutions of interest for future CPU routines need to be in CPU memory afterwards. By utilizing certain techniques this memory copy can be performed in parallel with other computations, but this is not always possible. For this reason, it might sometimes be worth performing serial computations on the GPU rather than on the CPU if this avoids a large memory copy. One should also keep in mind that one large transfer is faster than several small transfers and therefore strive to combine several small transfers into one large.

A question when optimizing code is: how long time will the optimization take versus the speedup gained? This is a hard question to answer. In [8], the authors claim that the performance increase by systematically tuning the parameters for the specific application as compared to heuristically settings is usually about 20%.

2.4.7 Some limitations and advantages

CUDA is a proprietary language of Nvidia, which comes with both drawbacks and advantages. The evolution of the language is entirely in the hands of Nvidia and as of today CUDA is only supported for Nvidia GPUs. The benefit, though, is that it can be streamlined for those GPUs without having to consider differences with other manufacturers. CUDA implementations have been reported informally on the web to be faster than other GPGPU languages such as OpenCL. However, if this is due to differences in optimization, hardware or the language/compiler is hard to know, and

this should only be taken as an indication. Another advantage increasing productivity, is that CUDA comes with many existing libraries, such as the math libraries cuBLAS, cuSPARSE, cuFFT, cuRAND, and Thrust. There is also a broad community of users.

A disadvantage with being proprietary, is that no one knows what Nvidia decides to do with CUDA in the future.

2.4.8 Code and optimization example: A vector sum

There are differences when writing code for serial programs for CPUs and massively parallel programs for GPUs. This section is intended to illustrate algorithmic differences, illustrate CUDA code, and highlight a few considerations for writing optimized CUDA kernels. The illustration will consist of writing and a few steps of the optimization, of a vector sum routine in CUDA. This section is not required to be understood to read the rest of this thesis, it is rather intended for those unfamiliar with CUDA interested in having a glance at what it looks like and how it works.

In computer science a vector sum on a parallel architecture is often done using a reduction algorithm. The problem is divided into smaller and smaller subproblems and eventually solved once small enough. A similar reduction as will be illustrated here, can be used to find the maximum or minimum of a vector. The example is highly inspired by the slides of a presentation by Mark Harris [16]. The reason for choosing a vector sum as an example is its simple problem description, and that such a routine was implemented to calculate solver errors in the final Navier-Stokes implementation. Observe that some assumptions have been made on the input to avoid complex code and handling exceptions. Such simplifications are that the input vector size is a power of two, and that some parameters are assumed to be known at compile time.

On a serial machine a routine for calculating the sum of a vector is trivially implemented, see Algorithm 1.

Algorithm 1 Implementation of a serial vector sum.

for $i = 1 \rightarrow n$ **do**
  $sum+ = vec[i]$
end for

The same algorithm implemented in CUDA would not yield any parallelism, instead a reduction approach will be taken. The problem can be solved using a Divide & Conquer algorithm, where the splitting part is trivial and the combining part is just combining the numbers until only one remains. Each leaf of the reduction tree will correspond to the computation performed by one of its child threads. Such an algorithm could look like Algorithm 2.

In CUDA a similar routine based on the Divide & Conquer approach for computations could look like Algorithm 3. For readers who have not seen CUDA code before, a lot is new and it might be hard to grasp all details. Recall that a kernel is launched with a number of blocks where each block can consist of up to thousands of threads. Each thread being launched by the kernel evaluates the body of the function. The __global__

Algorithm 2 Pseudo code for Divide & Conquer vector sum algorithm.

```
function reduction( data[ ], nElem)
if nElem == 2 then
  return data[1]+data[2]
else
  return reduction(data[1...nElem/2],nElem/2)+reduction(data[nElem/2+1,nElem],nElem/2)
end if
```

keyword, is what defines that the function is a CUDA kernel. threadIdx.x, blockIdx.x and blockDim.x are built in variables for keeping track of, and numbering, the threads and blocks inside a kernel. threadidx.x is the thread number (ID) within a block, blockIdx.x is the block number (ID) of that thread, and blockDim.x is the number of threads per block. The keyword __shared__ is what defines shared memory. Recall that through shared memory, threads within a block can communicate. The communication is done together with the __syncthreads() function, which says that before any of the threads in the same block can continue, all thread in that block must have reached the __syncthreads() mark. THREAD_PB is just a hard coding of the number of threads per block, done for simplicity but not necessary.

The essence of the algorithm is that all threads within a block reads an entry of the vector to shared memory. Consecutive blocks of threads reads consecutive elements of the input array to its shared memory. Each block of thread then performs a Divide & Conquer type of operation, written explicitly in form of a for-loop, on the elements stored in its shared memory. First, every second thread computes the addition of the value stored at the same position as its ID in the input array and the element after it, and stores the result at the position of its ID. Then, every fourth thread adds its element and the element two steps away, and stores it at its own position, and so on. Eventually, each block will have computed the sum of the elements loaded into its shared memory and stores the data in the position of the output array corresponding to its block ID. After the kernel has completed, the output array will consist of as many elements as there were blocks launched by the kernel. That is, if 512 blocks were launched by the kernel, then the output will consist of 512 elements. Thus, to complete the vector sum, the kernel has to be invoked several times with the computed output as the new input. This first algorithm illustrates the huge difference between the serial implementation and the parallel CUDA implementation. Not only does the CUDA implementation require a parallel algorithm, it is also bound by that communication between threads can only occur between threads within a block.

The above algorithm can be improved by observing how the threads are accessing data from the shared memory in the Divide & Conquer operation. The if statement causes diverging branches of the threads, something we wish to avoid, since neighboring threads are not performing the same operations. Further, the access from shared memory is not being coalesced, thereby causing bank conflicts. We would rather like to have consecutive threads access consecutive elements in shared memory. Luckily the access pattern is

Algorithm 3 Naive implementation of vector sum in CUDA C.

```
__global__ void vecSum1( int nElem, int[] iData, int[] oData ) {
int tID = threadIdx.x+blockIdx.x*blockDim.x;
int localTID = threadIdx.x;
__shared__ int s_oData[THREAD_PB];
oData[localTID] = 0;
if  tID < nElem  then
  s_oData[localTID] = inData[tID];
  __syncthreads();
  for int i = 1; i<blockDim.x; i = i*2 do
    if (localTID % 2*i) == 0 then
      s_oData[localTID] += s_oData[localTID+i];
    end if
    __syncthreads();
  end for
  if localTID==0 then
    oData[blockIdx.x] = s_oData[0];
  end if
end if
}
```

known beforehand and can be done such that consecutive threads access consecutive elements in shared memory. Instead of letting individual threads add together consecutive elements, consecutive threads should access consecutive elements for addition. In the next step, half as many threads adds elements from one fourth to half of the shared memory array and so on. Also note that only half of the threads are active, as threads with odd ID's remain idle. This can be resolved by loading twice the number of elements into the shared memory for each block, thereby launching half as many blocks when launching the kernel. These issues are resolved in the improved algorithm, Algorithm 4.

A further step to improve performance consists of unrolling loops in different levels, in this way reducing instruction time overheads. A clever way of doing this is noting that threads within the same warp execute their instructions simultaneously and that the size of a warp is 32, avoiding the need to use __syncthreads() in addition to the unrolling. Incorporating this change into the vector sum algorithm results in an algorithm like Algorithm 5. In the algorithm a __device__ function has been introduced, which is a function callable from kernels only. These are handy for structuring and reusing code. Often the body of these functions are directly inserted in the place where they are called by the compiler. The vector sum algorithm can further be improved by letting each thread calculate several entries from the input vector, in this way reducing the number of blocks being launched when the kernel is invoked. This and further optimizations are, however, not treated further here.

The three different algorithms are successively more effective for performing the

Algorithm 4 Slightly optimized implementation of vector sum in CUDA C, making consecutive threads perform more work together.

```
__global__ void vecSum1( int nElem, int[ ] iData, int[ ] oData ) {
int tID = threadIdx.x+blockIdx.x*(blockDim.x*2);
int localTID = threadIdx.x;
__shared__ int s_oData[THREAD_PB];
oData[localTID] = 0;
if  tID < nElem  then
   s_oData[localTID] = inData[tID];
   if tID+blockDim.x<nElem then
      s_oData[localTID]+=iData[tID+blockDim.x]
   end if
   __syncthreads();
   for int i =blockDim.x/2; i>0; i/=2 do
      if localTID < i then
         s_oData[localTID] += s_oData[localTID+i];
      end if
      __syncthreads();
   end for
   if localTID==0 then
      oData[blockIdx.x] = s_oData[0];
   end if
end if}
```

computation. The timings for a vector sum of a vector of $2^{23} = 8388608$ elements for the three different algorithms can be seen in Figure 2.1. The algorithm after the third optimization step is almost four times faster than the naive implementation.

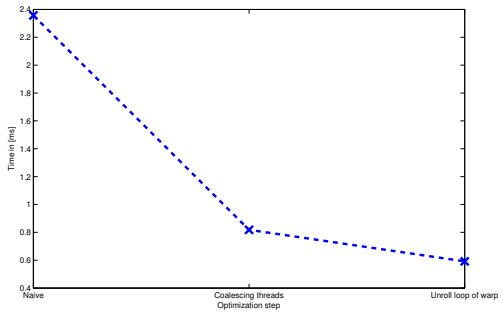


Figure 2.1: Timings for the three different optimization steps of the vector sum. The vector consisted of $2^{23} = 8388608$ elements.

Algorithm 5 Further improved implementation of a vector sum in CUDA C, a first step of loop unrolling.

```
__device__ void unrollWarpLoop( volatile int[ ] s_oData, int localTID) {
s_oData[localTID]+=s_oData[localTID+32];
s_oData[localTID]+=s_oData[localTID+16];
s_oData[localTID]+=s_oData[localTID+8];
s_oData[localTID]+=s_oData[localTID+4];
s_oData[localTID]+=s_oData[localTID+2];
s_oData[localTID]+=s_oData[localTID+1];
}

__global__ void vecSum1( int nElem, int[ ] iData, int[ ] oData ) {
int tID = threadIdx.x+blockIdx.x*(blockDim.x*2);
int localTID = threadIdx.x;
__shared__ int s_oData[THREAD_PB];
oData[localTID] = 0;
if  tID < nElem  then
   s_oData[localTID] = inData[tID];
   if tID+blockDim.x<nElem then
      s_oData[localTID]+=iData[tID+blockDim.x]
   end if
   __syncthreads();
   for int i =blockDim.x/2; i>32; i/=2 do
      if localTID < i then
         s_oData[localTID] += s_oData[localTID+i];
      end if
      __syncthreads();
   end for
   if localTID<32 then
      unrollWarpLoop(s_oData,localTID);
   end if
   if localTID==0 then
      oData[blockIdx.x] = s_oData[0];
   end if
end if
}
```

2.5 Parallel programming languages and GPGPU platforms other than CUDA

In this section an overview of other parallel programming languages, models, API's and hardware accelerators will be given. First common CPU oriented models will be explained, followed by many core alternatives.

MPI, an abbreviation for Message Passing Interface, is widely used by researchers and engineers for computation on scalable clusters. MPI is targeted for use when several compute nodes are being used, which do not share a memory. However the nodes themselves could be of shared memory architectures. "Messages" are passed to share data and communicate between nodes. MPI can be used together with CUDA GPUs.

OpenMP is a model building on pragma compiler directives for parallelization. It is designed for parallel programs on single compute nodes, i.e. shared memory architectures. Given specifications from the programmer, the compiler will automate the parallelization. It is mainly targeted for shared memory systems with several processors, such as multicore CPUs. OpenMP has a difficulty of arbitrary scaling to many cores, due to thread management and hardware specifics [8].

Pthreads (POSIX threads) is, like OpenMP, aimed at providing parallelism for shared memory architectures. In PThreads, the programmer is responsible for thread synchronization and management, giving him/her higher flexibility than OpenMP.

OpenCL was initially a project from Apple, but is now an open source project supervised by the Khronos group (also supervising OpenGL). OpenCL is similar to CUDA, but supports graphics cards from multiple vendors as well as multicore CPUs from several vendors. OpenCL applications are compiled for the specific target architecture at runtime. The goal of OpenCL is to create an industry standard for parallel programming, being able to run the same code on different computing devices.

DirectCompute is Microsoft's equivalent to CUDA and OpenCL for GPGPU programming, more closely linked to the DirectX graphics API. A similar platform from Microsoft is C++ AMP, which builds on DirectCompute but provides an interface for C++ developers.

OpenACC is a programming standard based on compiler directives, similar to that of OpenMP, aimed at simplifying programming of heterogeneous CPU, GPU and other accelerator programs. It is developed by CRAY, CAPS, Nvidia and PGI. Compiler directives are used to specify the parallelism, the compiler then writes the necessary code to transfer data and performing other necessary operations and port that part of the code to GPU/accelerator code.

Intel MIC (Many Integrated Core) Architecture, with the brand name known as Intel Xeon PHI, is Intel's answer on the increasing trend of GPUs in the High Performance Computing industry. The accelerators are used as co-processors to the CPU, yielding high memory bandwidth, high processing power, and low energy consumption, similarly to GPUs.

Chapter 3

Previous work

Applications and problems suitable for GPU computing should be highly parallelizable, be carried out over large amounts of data, and individual data elements should have minimal dependency. The interest for GPU accelerated engineering and scientific applications has been large for the last years. Applications within many different fields have been accelerated by GPUs, such as biology and medicine research, seismic calculations, engineering computations, gaming, image processing, and finance. In this chapter, what have been done on GPUs in the fields of sparse linear algebra solvers and computational fluid dynamics relevant to this thesis work will be presented.

Some terms introduced in this chapter will be described in later chapters. The uninitiated reader can therefore find an interest in reading this chapter again or after reading the chapters following.

3.1 Sparse Linear Algebra on GPUs

Numerous articles and reports have been published where linear algebra routines have been accelerated on GPUs. Both sparse and dense linear algebra have been studied. Dense linear algebra implementations show promising speedups, sparse linear algebra implementations report speedups but not as high. Sparse linear algebra is a corner stone in many scientific applications. In general in those involving discretization of PDEs such as fluid mechanics and structural mechanics. This motivates the high interest in literature, especially so as many linear algebra operations are inherently parallel.

Efficient SpMV implementations have been studied among others by [17, 18, 19]. In [17], efficient implementations of the SpMV operation are investigated. Implementation and performance of several sparse formats are investigated and compared to SpMV implementations on CPUs. The HYB-sparse format is the best general format for the majority of matrices tested. The GPU implementations outperform CPU implementations with up to an order of magnitude if the matrices are decently sparse. It is also mentioned that the usage of texture memory for the vector x increases performance with about 25 %. This is achieved by data reuse since texture memory is cached and x can be reused. In [19], different implementations of the SpMV operation in the CSR sparse format is

studied exclusively. The study reports an optimized SpMV in CSR format achieving comparable results to Nvidia's cuSPARSE library's SpMV operation for the HYB sparse format.

Preconditioned iterative solvers accelerated on the GPU are studied in [18]. All parts of the preconditioners and solvers are evaluated for implementation on the GPU, including the SpMV operation and solution algorithms for triangular systems. They find that the GPU SpMV implementations are superior to CPU implementations. For GPUs, the JAD sparse format is found to be faster than the CSR format. The speedup against a parallel CPU implementation is up to a factor of 5. The GPU accelerated triangular solve operation, outperforms the CPU implementation by a factor of up to five if a technique called level scheduling is being used. Incomplete Cholesky preconditioned CG was reported to give moderate speedups over a CPU equivalent, while ILUT preconditioned GMRES was reported to yield a speedup of three to four. The least square polynomials preconditioner was also tested together with the Conjugate Gradient iterative solver (CG). Speedups of up to seven to GPU Incomplete Cholesky preconditioned CG was reported, but for some of the matrices tested they performed equally well.

Efficient implementations of the incomplete-LU and Cholesky preconditioners on the GPU are studied in [20]. The preconditioners are parallelized using a technique called implicit orderings. The authors report speedups of an average of two when incorporated with solvers compared to CPU implementations.

In [21] kernel fusions of BLAS routines are discussed, meaning that several BLAS operations are combined into one operation. Reporting speedups of up to 2.61 when performing the same operations sequentially with cuBLAS. The benefit of kernel fusion lies in reducing memory fetches from off chip memory, in the GPU case global memory.

In [22], a simple multigrid method for a pressure Poisson equation for use in graphics oriented fluid flow simulations was implemented on the GPU. The speedup over a CPU multigrid method was up to above 50, while the speedup to a GPU implemented preconditioned CG was about eight. The implemented method was reported not to be able to handle complex domains.

In [23], a geometric multigrid method for unstructured finite element grids based on SpMV operations, which could handle complex domains, was developed. Reported speedups when using the GPU over a parallel CPU multigrid implementation was between six and ten for large grids.

3.2 Computational fluid mechanics on GPUs

The idea of GPU accelerated fluid simulations were present early in the GPGPU community. The primary applications initially was computer graphics. Navier-Stokes solvers were implemented on GPUs before both CUDA, and the Brook and SH programming models. Many of the graphics fluid flow simulation applications are based on Stam's method [24], producing nice real time effects, but not being accurate enough for scientific applications.

GPU implemented solvers of the Euler equations appear in the literature. The Euler

equations is a simplified version of the full incompressible Navier-Stokes equations, where the viscosity is assumed to be negligible. In [25], an Euler solver for a discretization on an unstructured grid reports to give an average speedup of about 1.5 for double precision. In [26], a 3D Euler Solver using the finite volume method is used to solve inviscid flow aimed at graphics simulations. The reported grids were fairly small (< 400000 cells). A tiling algorithm is used for assembly routines to decrease the memory fetches from global memory. A block of threads is iterating through blocks of spatial nodes through one of the domain dimensions. A speedup of sixteen times were reported using CUDA compared to a serial CPU implementation.

In [27], an incompressible Navier-Stokes solver was implemented on single and multi-GPU systems (up to 4 GPUs), and it was compared to a serial CPU implementation. For a single GPU, speedups of up to 13 and 33, respectively, for two different CPUs, were reported. Using two GPUs a speedup of less than two compared to a single GPU system was reported. For four GPUs, a speedup of up to three times was reported. The method implemented was an explicit time projection method, first solving the momentum equation uncoupled from pressure, then correcting it through a pressure Poisson equation solved using Jacobi iterations as iterative solver.

In [28], a double precision CFD code using the Boussinesq approximation was developed, solving incompressible flow of Rayleigh-Bernhard convection. An explicit Adams-Bashforth discretization is used for the temporal term and a regular staggered grid was used for the computational domain. A GPU multigrid method for simple geometries was implemented to solve the pressure Poisson equation. For large enough grids, a speedup of about 8 times compared to an OpenMP and MPI threaded CPU implementation using 8-cores was reported.

In [29], some CFD utility functions (block-tri diagonal solver, FVM maximum time step kernel, Euler fluxes) were implemented on a GPU using CUDA. Some speedups were reported compared to an OpenMP threaded CPU implementation. However, the bottleneck in CPU-GPU memory transfer was encountered, thereby limiting the use of only kernel accelerated applications.

Lattice Boltzman Methods (LBM) is another method of interest for solving fluid problems. It is amenable for massive parallelism due to its parallel nature. LBM is based on cellular automata. The fluid is modeled as particles on a discrete lattice of cells. Various GPU implementations of LBM exist in the literature. An early implementation of LBM on GPU, developed before the time of CUDA and OpenCL was the work done by [30], where an LBM is implemented on a GPU cluster. The article is of historical interest as well. A more recent LBM implementation on the GPU can be found in [31], where a speedup of around 100 over a CPU implementation is achieved for the Lid-Driven cavity case.

Chapter 4

Sparse matrix-vector kernel and sparse matrix formats

The Sparse Matrix-Vector kernel, denoted SpMV, is the operation usually defined by

$$y = \alpha Ax + \beta y, \tag{4.1}$$

where A is a matrix, and x and y are vectors. It accounts for up to half of the time required for many iterative methods. An efficient implementation of the SpMV kernel is thus of great importance for effective implementation of iterative methods. The performance of the SpMV kernel depends on the sparse matrix format used for the matrix. The sparse matrix format is also of importance when assembling the matrix, i.e. computing the matrix coefficients from the discretized Navier-Stokes equations. When assembling, it is desirable with an intuitive format with good access alignment and locality of data. Ideally, one would like to assemble directly into the format being used for the SpMV operation in the iterative methods. If this is not possible for the fastest format, a transfer of format might be the best choice. Another consideration is if theSpMV implementations available in cuSPARSE, or in a non-restrictive open source library, is fast enough; or if an implementation specifically for this thesis work should be considered.

The number of floating point operations of the SpMV, omitting the generalization constants α and β, are two times the number of non-zero elements in A. One multiplication and one addition per element in A. Each non-zero element in A is only used once while each element of x could be used several times. The number of memory reads required for one multiplication and one addition is at most two, fetching the x element and the non-zero element of A. Ideally, each element of x would only be needed to be fetched once from global memory and then stored in the cache. Depending on the effectiveness of caching and the number of non-zero elements in A, the floating point operation:memory access ratio lies between 1:1 and 2:1. Consequently the SpMV operation is memory intensive and likely memory bound. This is the main issue limiting the performance of SpMV on parallel architectures. The operation is of low arithmetic complexity and has irregular memory accesses. To achieve an efficient implementation of the SpMV operation, the memory access pattern is of uttermost importance and consequently the

sparse format of the matrix. Recall that the peak bandwidth of the GPU cannot be achieved unless coalesced memory accesses from global memory is being used.

Matrices resulting from discretization of engineering problems, or more specifically those resulting from FEM, Finite Volume or Finite Difference discretizations, are often extremely sparse. The efficiency of computation of these methods largely depends on local dependence for computing properties in discretized nodes, even though exceptions exist. This is also the case for CFD matrices, which commonly are based on a Finite Volume or a Finite Element discretization. Matrices commonly have an order of millions of rows and only on the order of tens of non-zeros per row. Normal matrix storage would require storing $O(N^2)$ numbers for an NxN matrix, while the actual number of unknowns is only $O(N)$. This motivate the usage of sparse storage formats.

4.1 Sparse matrix formats

The main difference between different sparse matrix formats is their memory requirements and efficiency on different sparsity patterns. Some can be efficient on regular patterns and other on those with many non-zero elements per row. First, several popular sparse matrix formats is briefly described, alongside with their benefits and drawbacks. Then, timing measurements of different open source implementations of the SpMV operation will be detailed. The interested reader can find a more detailed description of sparse matrix formats in [32], [17] and [18] among others.

4.1.1 COO format

The COOrdinate, COO, format is a straightforward general implementation of a sparse matrix. Three vectors are stored, each being of length equal to the number of non-zero elements of the matrix. The vectors the column indices, row indices and values, respectively, for the non-zero elements. This format stores some redundant information. The CSR format, explained next, is a more compact description of the matrix.

4.1.2 CSR format

In the Compressed Sparse Row, CSR, format one uses the redundant information that after one row the next follows. It results in that the row storage vector can be of the size $M + 1$ for an $M - by - N$ matrix. The column vector stores the column index of each non-zero element and the value vector stores the value of the non-zero elements. The row vector stores the position in the value vector for the first non-zero element in each row. It also stores an additional $M + 1$st element, which is the total number of non-zero elements in the matrix. The CSR format is a general purpose-, intuitive-, and popular sparse format. Other advantages is that the non-zeros per row can be trivially computed and that retrieving specific elements can quickly be done.

4.1.3 DIA format

DIA is not a general purpose format, but is constructed specifically for matrices consisting of a number of diagonals. Two arrays are needed for storage in the DIA format, one containing the offset of each diagonal from the main diagonal and one containing the values. The diagonals which are not the main diagonal, and thus shorter, are padded with arbitrary elements. It is this padding which makes this format unsuitable for arbitrary matrices. The implicit knowledge of the elements row and column indices through the diagonal offsets, makes the memory fetches efficient and memory requirement low on matrices containing only a few number of diagonals, for example those resulting from stencil discretization.

4.1.4 ELLPACK (ELL) format

The ELL format is another format which is unsuitable for general matrices, but can be efficient on decently regular matrices. Assume that an $M - by - N$ matrix have a maximum number of non-zeros for any of its row being n, then the ELL format consists of two $M - by - n$ matrices. One stores the nonzero values and the other the column index for each corresponding value. For rows containing less than n non-zeros, the extra entries are padded by arbitrary values. The ELL format is suitable for unstructured matrices for which the number of non-zeros per row are similar for all rows. The format has a beneficial memory access pattern for vector architectures, such as the GPU.

4.1.5 HYB format

The Hybrid (HYB) format is a format being a hybrid of the general COO format and the faster but more specific format ELL. Rows of the matrix containing less non-zeros than a specified value, k, are stored in the ELL format while those containing more elements are stored in the COO format. The format exploits that there are limitations on the number of vertices (nodes) with exceptionally high degree (connected to many other nodes) on discrete meshes of low dimensions. Reference [17] reports that empirically on well structured meshes the ELL format is about three times as fast as the COO format, with an empirical value of k chosen such that one third of the matrix rows contains k or more non-zero elements.

4.1.6 JAD format

The Jagged Diagonal format (JAD) is a general purpose sparse format. The format requires three vectors, one containing the values, one containing the column positions of the non-zero elements and one containing the start position of each row. First the rows of the matrix are sorted by the number of non-zero elements in decreasing order. Now the value matrix is built up by a sequence of JAD vectors, the number of JAD-vectors equals the maximum number of non-zeros per row. The first JAD-vector consists of the first non-zero element per row of the sorted matrix. The second consist of the second non-zero element per row and so on. The column vector contains the column position

of the elements in the value vector, and the index vector contains the position of the first element in each JAD. Now if one assigns a thread for each row of the matrix in the sorted order, then the first thread will access the first element in the JAD vector, the second, the second element and so on. Thus, the memory fetches can be coalesced. It is advantagous for GPU architectures since the memory is aligned for contiguous threads.

4.2 SpMV performance

Different SpMV implementations for different sparse formats were tested to evaluate suitable formats for computation. Implementations in CUDA Iter [33], Paralution [34] and Nvidia's cuSPARSE library were evaluated. The aim of the test was partly to see which formats were best suited for the specific matrices and partly to see if there was any need to implement an SpMV kernel for the purpose of this thesis. Since the result of this thesis work might be used in commercial code in the future, the use of restricting open source licenses would like to be avoided.

Double and single precision timings for an SpMV operation of a fluid mechanics finite volume discretized momentum matrix without grid refinements can be seen in Table 4.1. Timings for a matrix with a refined grid can be seen in Table 4.2. Specifications of the matrices can be found in Table 5.1. It is clearly seen that the sparse matrix format matters. The Paralution COO format is up to 3 times slower than the best Paralution format, while the ELL and HYB formats are consistently faster than CSR. For CuSparse the HYB format is about twice as fast as CSR. Nvidia's HYB format is the fastest for both matrices in double and single precision. The reason why CUDA Iter is slower is unknown, but might be due to bad optimization of recent GPUs as the latest update of the code was no later than 2011. The conclusion is that the HYB implementation in the cuSPARSE library is fastest. It is the implementation used for the SpMV operation in the iterative methods described in the following chapter.

An interesting property which will be discussed further on, is that the timing difference between single and double precision is slightly less than a factor two. As double precision floating point numbers requires 8 byte while single precision requires 4 bytes, this indicates that the hypothesis that the SpMV kernels are memory bound is likely correct.. The reason why the differences are not exactly a factor two can be that some of the sparse vectors, such as row and column pointer in CSR format, are integers, which are of the same size regardless of if single or double precision is being used for the other vectors.

Table 4.1: Timings for the SpMV operation in double and single precision for different sparse formats and implementations. The operation is evaluated on the momentum1M matrix.

	CuSPARSE		**Paralution**		**CUDA Iter**	
Format	SP [ms]	DP [ms]	SP [ms]	DP [ms]	SP [ms]	DP [ms]
CSR	1.30	2.30	1.24	2.08	-	6.49
HYB	0.67	1.15	1.01	1.83	-	-
JAD	-	-	-	-	-	1.57
ELL	-	-	0.63	1.13	-	-
COO	-	-	2.49	3.17	-	-

Table 4.2: Timings for the SpMV operation in double and single precision for different sparse formats and implementations. The operation is evaluated on using the momentumR1.7M matrix.

	CuSPARSE		**Paralution**		**CUDA Iter**	
Format	SP [ms]	DP [ms]	SP [ms]	DP [ms]	SP [ms]	DP [ms]
CSR	2.30	3.99	2.15	3.64	3.23	6.49
HYB	1.36	2.32	1.69	2.72	-	-
JAD	-	-	-	-	-	3.29
ELL	-	-	1.81	2.79	-	-
COO	-	-	4.61	6.26	-	-

Chapter 5

Iterative methods and solvers

Linear systems arise in discretization of many engineering and scientific applications, specifically those involving finite volume and finite element discretization. Solving such systems, $Ax = b$, can be done by calculating the inverse directly and applying it to both sides of the equation resulting in $x = A^{-1}b$. This involves calculating the inverse of A, using a so called direct method, which generally is expensive with respect to time and memory for large systems. If the system is to be solved multiple times with the same matrix but different right hand sides, the process can be speeded up by factorizing the matrix, in which the factorization might be expensive but where the solve phase afterwards is relatively cheap.

For non-linear PDE's such as the Navier-Stokes equations the matrices need to be rebuilt often, eliminating the reusable benefit of factorization. Often a more efficient way to solve such a system is by using an iterative method. In these methods, the inverse A^{-1} is never calculated explicitly, one rather uses an algorithm that iteratively converges towards the solution x. Three popular such methods are the Conjugate Gradient method (CG), the Generalized Minimal Residual Method (GMRES) and Bi-Conjugate Gradient Stabilized method (BiCGStab). These methods all fall under the category of iterative methods called Krylov subspace methods.

In the following a brief description of the implemented methods and their underlying idea, together with limitations and advantages, will be presented. For the interested reader, a good and extensive textbook is [32], another is [35], an application based shorter book with details and practicalities of implementations is [36].

5.1 Krylov subspace methods

The Krylov subspace methods search for the solution of the linear system $Ax = b$ in a certain subspace of the whole domain called the Krylov subspace. The methods try to project the solution onto the Krylov subspace. The methods are part of a larger family of methods called projection methods. The Krylov subspace is spanned by vectors of polynomials of A of the form $p(A)r$, where $r = Ax - b$ is the residual. The iterative methods then projects the solution onto a Krylov subspace of increased dimension, in

this way converging towards the solution in each iteration. Many of the commonly used iterative methods fall within this Krylov subspace family of methods.

Before diving deeper into the different Krylov subspace methods, a brief overview of the main idea of projection methods will be given. By utilizing a projection method for solving $Ax = b$, A is n x n, we seek to find an approximate solution of the system in a subspace of the actual solution space $\mathbb{R}$. Let us define $\mathcal{K}$ as our search space and $\mathcal{L}$ as our subspace of constraints, both being subspaces of $\mathbb{R}$ of size m. The way to proceed to find an approximate solution $\tilde{x}$ in our search space $\mathcal{K}$, is by iteratively building them up, ensuring that $\tilde{x} \in \mathcal{K}$ and that the residual vector of the approximate solution is orthogonal to our subspace of constraints, i.e.

$$\text{Find } \tilde{x} \in x_0 + \mathcal{K} \quad \text{s.t } b - A\tilde{x} \perp \mathcal{L}. \tag{5.1}$$

There are two major classes of projection methods. In orthogonal methods $\mathcal{L} = \mathcal{K}$. In oblique methods $\mathcal{L}$ and $\mathcal{K}$ are different and may even be unrelated to each other. Krylov subspace methods can belong to both classes, what defines them is that the search space is the Krylov subspace

$$\mathcal{K}_m = span\{r_0, Ar_0, ..., A^{m-1}r_0\}, \tag{5.2}$$

where $r_0 = b - Ax_0$ is the initial residual vector.

In the iterative methods the dimension of the Krylov subspace is increased by one at each iteration. For certain choices of the constraint subspace the methods can be proven to converge at latest after n iterations, n being the dimension of the matrix. What differs between the different Krylov subspace methods is how the constraint subspace $\mathcal{L}$ is chosen. Two common choices resulting in CG and GMRES, respectively, are $\mathcal{L} = \mathcal{K}_m$ and $\mathcal{L} = A\mathcal{K}_m$. The underlying idea for building an orthogonal basis of the Krylov subspace is Arnoldi's algorithm, see [35] or [32], this idea is used for many of the Krylov subspace iterative methods.

Before the specific Krylov subspace methods implemented in this thesis is described, an overview of preconditioning, often used together with iterative methods, is given.

5.1.1 Preconditioning explained briefly

The convergence rate of Krylov subspace iterative methods depends on the condition number of the coefficient matrix, A, in turn related to the spectrum (range of eigenvalues) of A [35]. If the coefficient matrix is ill conditioned, the number of iterations can be large. In these cases, one would like to reduce the condition number of the coefficient matrix. Instead of considering the system $Ax = b$, let us consider the equivalent system

$$M^{-1}Ax = M^{-1}b = \widetilde{A}x \Leftrightarrow \widetilde{b} \tag{5.3}$$

where M is a matrix of the same size as A. The condition number of $\widetilde{A}$ can be much better than that of A and can result in significantly faster convergence. Transforming systems in this way is called preconditioning. The least number of iterations would be achieved if $M^{-1} = A^{-1}$, but then the method is just a direct solver, on which it is

extremely expensive to perform the matrix inversion M^{-1}. The cheapest inversion would be achieved by choosing $M = I$, being the identity matrix, but would cause no difference for the condition number or number of iterations until convergence. The desirable choice is to have a preconditioner which is cheap to invert (or perform an inversion operation), while still resembling the main features of A.

There exist different ways to apply a preconditioner. Left preconditioning is the technique described above, right preconditioning is given by

$$AM^{-1}\widetilde{x} = b, \quad \widetilde{x} = Mx. \tag{5.4}$$

A third variant exist if the preconditioning matrix is given in a factorized form $M = M_L M_R$,

$$M_L^{-1} A M_R^{-1}\widetilde{x} = M_L^{-1} b, \quad \widetilde{x} = M_R x. \tag{5.5}$$

Using preconditioners might affect symmetry of the resulting system if the original matrix A is symmetric. However breaking the symmetry does not necessarily affect the convergence rate assuming M is symmetric positive definite [35]. There also exists preconditioners where no explicit matrix M is formed, the preconditioner is instead an operation, $s(A)$, of the matrix A. The least square polynomials preconditioner implemented in this work is such a preconditioner.

5.1.2 Conjugate Gradient Method (CG)

The Conjugate Gradient Method (CG) is a popular choice for solving systems where the coefficient matrix is symmetric and positive definite. As already mentioned, the constraint space for the CG algorithm in the m:th iteration is $\mathcal{L}_m = \mathcal{K}_m$. The method consists in generating a sequence of vectors converging towards the true solution vector. The vectors are generated by searching a distance along a search direction, resulting in vectors being conjugate to each other, $x_i A x_{i+1} = 0$. Associated with every vector x_i is an associated residual vector r_i. The preconditioned CG algorithm can be found in general textbooks such as [32] or [37] and is outlined in Algorithm 6 below. Note that choosing $M = I$ is equivalent to the non-preconditioned CG algorithm.

Using a preconditioner the symmetry of the resulting matrix $\widetilde{A}$ might be destroyed. However, this is not an issue if the matrix is symmetric and positive definite, as was mentioned in the previous section. In this case, it is not necessary to split the preconditioner. It can be proven that splitting the preconditioner or using a modified algorithm of CG results in the exact same iterates [32].

In the CG Algorithm, Algorithm 6, p is the search direction vector and α the distance to search along this vector for each iteration. The p_{i+1}'s and r_{i+1}'s of the CG algorithm are orthogonal to all previous p_i's and r_i's. Moreover the choice of α and β minimizes

$$(x_i - \hat{x}, A(x_i - \hat{x})) \tag{5.6}$$

in the affine space $x_0 + \mathcal{K}_m$ [36], where $\hat{x}$ is the exact solution. In general, this minimum is only guaranteed to exist if A is symmetric and positive definite. In this case the method

Algorithm 6 Preconditioned CG algorithm.

Set initial guess $x_0 = 0$
Compute $r_0 = b - Ax_0$
Solve $Mz_0 = r_0$
Copy $p_o \leftarrow z_0$
for $i = 1$...convergence **do**
  $\alpha_i = \frac{(r_i, z_i)}{Ap_i p_i}$
  $r_{j+1} = r_j - \alpha_j A p_j$
  $x_{j+1} = x_j + \alpha_j p_j$
  $z_{j+1} = M^{-1} r_{j+1}$
  $\beta_j = \frac{(r_{j+1}, z_{j+1})}{(r_j, z_j)}$
  $p_{j+1} = z_{j+1} + \beta_j p_j$
end for

converges in at most n steps where n is the size of the matrix [37]. There exist upper limits for the convergence of the i:th iteration. Suppose that A is symmetric and positive definite, and be aware that $||x||_A^2 = (x,Ax)$. Note also that the spectral condition number of A is given by $\kappa_2(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}$, where λ is an eigenvalue of A. Then the following holds [37]

$$||x_i - \hat{x}||_A \leq 2\alpha^i \, ||x_0 - \hat{x}||_A \,, \quad \alpha_i = \frac{\sqrt{\kappa_2(A)} - 1}{\kappa_2(A) + 1}. \tag{5.7}$$

Specifically, for a typical second order elliptic partial differential equations $\kappa_2(A) = O(h^{-2})$, where h is the mesh length. Implying that the number of iterations for non-preconditioned CG will scale proportional to h^{-1} [36]. However, the actual convergence rate is usually faster than the worst case and depends not only on the range of the spectrum, but also on the disitribution of the spectrum. In general, the method converges fast in the norm $||.||_A$ if $k_2(A) \approx 1$.

The CG algorithm requires two inner products, three AXPY operations (defined by $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$), one matrix-vector product, and one preconditioner solve operation per iteration. The storage requirement is: one matrix (plus preconditioner matrix, if applicable) plus $6n$. The conjugate gradient method is attractive because of its relatively inexpensive inner loop (computational cost per iteration), and low memory storage requirements. Only four vectors need to be stored rather than all the krylov subspace vectors. The time required for solving Poisson's equation in 2D with CG is $\mathcal{O}(n^{3/2})$, where $n = N^2$ is the total number of nodes [35].

5.1.3 Generalized Minimal Residual Method (GMRES)

The Generalized Minimal Residual method (GMRES) was first proposed by [38] and is obtained by choosing the constraint subspace as $\mathcal{L}_m = A\mathcal{K}_m$. It can be proven, [32] or [35], that this choice of constraint search space, $\mathcal{L}_m$, minimizes the norm of the residual

vector in our Krylov subspace, $||b - Ax_i||$. That is,

$$x_m = argmin||b - Ax||_2, \quad \forall x \in \{\mathcal{K}_m + x_0\}. \tag{5.8}$$

Let us clarify the mathematical statement in words, GMRES finds the best solution in the Krylov subspace built up thus far in aspect of the residual. That is, the method minimizes the residual $r_k = b - Ax_k$ of the vectors in the Krylov subspace of dimension k. GMRES is a generalization of a method called MinRES, extending the applicability from symmetric matrices to general matrices.

There exist a few variations of the GMRES method, the main difference between them is how to perform the orthogonalization of Krylov subspace vectors. Commonly one either uses householder transformations or Gram-Schmidt orthogonalization. The former requires more work, is more stable and has increased convergence rates for ill conditioned systems, while the latter can utilize a higher degree of parallelism [36]. In the following the householder variant will be explained, which is the method implemented in this thesis work.

Each iteration of the method consists of finding a Krylov subspace vector, using an iteration of the Arnoldi method, orthogonalizing the vectors and then calculating the vector in the Krylov subspace $\mathcal{K}_m$ corresponding to the minimal residual as defined in (5.8). Updating the iterate x_i at each iteration i is expensive, however, this can be avoided using properties of the Hessenberg matrix resulting from Arnoldi's algorithm. The Hessenberg matrix can in each iteration be transformed to QR form using Given's rotations, from which the residual and the update of the iterate can be obtained. To obtain the iterate, a least squares problem need to be solved. On the other hand, this can be postponed until the iterate is needed explicitly. The interested reader is referred to [32] for derivations.

A drawback with GMRES is that one needs to store the whole Krylov subspace calculated thus far, requiring a significant amount of memory, and also increasing the work required per iteration. What one generally does is to restart the GMRES after a fixed number of iterations, with the current solution as input to the restarted GMRES. This decreases the memory requirements and deteriorates the convergence rate but works reasonably well in practice. This further motivates the use of a preconditioner together with GMRES. The algorithm used in the GPU implementation for the left preconditioned restarted GMRES method is given in Algorithm 7.

Algorithm 7 Left preconditioned restarted GMRES algorithm using householder transformations.

initial guess x_0
for $i = 1...$convergence **do**
  solve r from $Mr = Ax_0$
  $\beta = \|r\|_2$
  $V_{:,1} = r/\beta$
  for $j = 1...m =$ Iterations before restart **do**
    solve z from $Mz = AV_{:,j}$
    for $k = 1...j$ **do**
      $H_{k,j} = V_{:,k}z$
      $z = z - H_{k,j}V_{:,k}$
    end for
    $H_{j+1,j} = \|z\|_2$
    $V_{:,j+1} = z/H_{j+1,1}$
    Givens rotations to transform H to its QR factor, residual given by the transformation
  end for
  Compute y_m as $argmin_y\|\beta e_1 - Hy\|_2$ by solving upper triangular QR factor {Least square problem}
  Update $x_m = x_0 - Vy_m$
end for

In the algorithm V is an N x m matrix ,where N is the number of equations and m is the number of iterations before restart, H is an $(m+1)$ x m Hessenberg matrix generated as a step in the Arnoldi iteration. The least squares problem can be solved in a clever way by utilizing that H is a Hessenberg matrix, thereby using Givens rotations to transform H to QR-form. Also the triangular system need not be solved explicitly in every iteration step to calculate the residual. It is only solved before restarting the algorithm or after convergence. See [32] for implementation details. In the ideal case of non-restarted GMRES, the method will converge within n iterations, where n is the size of the matrix.

In the implemented GMRES algorithm in this thesis, all BLAS1 routines and SpMV operations are performed on the GPU. The Givens rotations are performed on the CPU, however no heavy memory transfers are necessary as the elements of H is calculated one per GPU operation and the Givens rotation is not expensive. The triangular solve is also performed on the CPU, but is not expensive to compute either, as will be seen in the results section. Thus, all expensive operations are easy to parallelize and performed on the GPU, making it well suited for GPU computation.

GMRES requires $i+1$ inner products, $i+1$ AXPY operations, one SpMV, and one preconditioner solve operation for the i:th iteration since restart. The storage requirements is that of the matrix plus that of the preconditioner matrix plus $(i+5)n$, for the i:th iteration since restart.

5.1.4 Bi-Conjuage Gradient Stabilized Method (BiCGStab)

The Bi-Conjugate Gradient Stabilized method (BiCGSTAB) is a variation of the CG method which, in contrast to CG, successfully can be applied to non-symmetric matrices. The CG method can only keep orthogonal residual vectors if the matrix A is symmetric. BiCGStab tries to remedy this, with the aim of extending CG to a method for general matrices without having to keep the sequence of vector iterates (required by GMRES among others). The idea behind BiCGStab is keeping track of two pair of residual and search vectors and constants, one for A and one for the transpose of A. The drawbacks of BiCGStab are that no global minimization is guaranteed, which can lead to irregular convergence, and that it occasionally can fail to find a solution [36]. The breakdown seems to occur rarely in practice. Its advantages is that it still posses the attractive features of CG of constant storage and a relatively cheap inner loop. The algorithm for the left preconditioned BiCGSTAB method is similar to that of [39] and is presented in Algorithm 8.

Algorithm 8 The left preconditioned BiCGSTAB method.

Initial guess x_0
Compute $r = b - Ax_0$
Set $p = r$
Set arbitrary $\hat{r}$
for $i = 1$...convergence **do**
  $\rho_i = \hat{r} * r$
  if i > 1 **then**
    $\beta = \frac{\rho_i}{\rho_{i-1}} \cdot \frac{\alpha}{\omega}$
    $p = r + \beta(p - \omega q)$
  end if
  Solve $M\hat{p} = p$
  $q = A\hat{p}$
  $\alpha = \frac{\rho_i}{(r,q)}$
  $r = r - \alpha q$
  Solve $Ms = r$
  $t = As$
  $\omega = \frac{(t,r)}{(t,t)}$
  $x = x + \alpha\hat{p} + \omega s$
  $r = r - \omega t$
end for

The BiCGStab algorithm requires four inner products, six AXPY operations, two SpMV operations and two preconditioner solve operations per iteration. The storage required is that required for the matrix and preconditioner matrix plus $10n$.

5.2 Preconditioners

5.2.1 Diagonal scaling

The simplest form of preconditioning is diagonal scaling, also known as Jacobi preconditioning, in which the preconditioning matrix is chosen as $D = diag(A)$. This can be implemented as a general preconditioner solving the preconditioned system

$$D^{-1}Ax = D^{-1}b, \tag{5.9}$$

using a Krylov subspace method. But a more computationally efficient implementation is to directly apply the diagonal scaling matrix D^{-1} on both sides of the system and then solve the resulting system $Bx = f$, where $B = D^{-1}A$ and $f = D^{-1}b$. Doing this might however create a non-symmetric matrix B from a symmetric matrix A. A more stable way, keeping symmetry in the transformation if any, is solving the transformed system

$$D^{-1/2}AD^{-1/2}u = D^{-1/2}b, \quad u = D^{1/2}x, \tag{5.10}$$

resulting in

$$Bu = f, \quad B = D^{-1/2}AD^{-1/2}, \; u = D^{1/2}x \; f = D^{-1/2}b. \tag{5.11}$$

After the iterative method has converged, the solution x can be retrieved from u by $x = D^{-1/2}u$. Diagonal scaling is efficient for diagonally dominant systems.

5.2.2 Incomplete LU factorization

In the incomplete LU factorization the term incomplete means that, for the factorization, some terms which would be non-zero in the complete factorization matrix, but are zero in the coefficient matrix, have been set to zero. Recall that storing all elements of a sparse matrix explicitly requires unfeasible amounts of storage. The incomplete LU factorization is a LU factorization $M = LU$ but with some elements in the factorization dropped to zero. Specifically, incomplete LU with zero fill in (ILU0), is the LU factorization where only elements at positions which are non-zero in the coefficient matrix are non-zero in the LU factors.

The ILU0 preconditioner was not implemented in this thesis work. Instead the existing implementation in Nvidia's cuSPARSE library was used for preconditioning the iterative solvers. The parallelization method in the implementation is divided into an analysis and one factorization phase being based on implicit orderings [20]. The implementation is reported to give an average speedup of 2.7 times for the factorization phase compared to MKL's ILU0 implementation. However, this number is not the whole truth, since to apply the precondtioner the incomplete lower and upper factors have to be computed. Typically a tridiagonal system has to be solved. The GPU tridiagonal system solver implementation used is reported to give an average speedup of 2 times compared to the MKL library [39].

5.2.3 Least square polynomials

Polynomial preconditioners falls into a category of preconditioners trying to approximate A^{-1} by an operation of the original matrix $s(A)$. In this way the preconditioner matrix M is never formed explicitly, and memory requirements are kept low. The preconditioned system to solve is $s(A)Ax = s(A)b$. For the Least Square Polynomial (LSP) preconditioner, the approximation of A^{-1} is by a polynomial in A. The polynomials $s(A)$ and A commute [32], meaning that left or right preconditioning makes no difference. It will be seen that forming the LSP preconditioner depends on estimating the spectrum of the coefficient matrix A, i.e. its largest and smallest eigenvalues, λ_{min} and λ_{max}.

The question of how to choose the polynomial $s(A)$ boils down to the fundamental desirable properties of a general preconditioner. Ideally, one would wish to have $s(A)A$ as close to the identity matrix as possible. A good and less expensive approximation is to make its eigenvalues similar to that of the identity matrix. Recall that the convergence rate for the CG algorithm is greatly dependent on the spectrum of the matrix, meaning that reducing the range of the spectrum will increase the convergence rate. The following problem formulation is used to define the least square polynomial $s(A)$ [32]: Let P_k be the space of polynomials of grade k and lower, and $[\alpha,\beta]$ the interval containing the eigenvalue range of A. Then, the aim is to find $s_{k-1} \in P_{k-1}$ minimizing

$$\|1 - \lambda s(\lambda)\|_w, \tag{5.12}$$

where w is some weight and the w norm is defined by $\|.\|_w = \sqrt{(.,.)}$ where

$$(p, q) = \int_a^b p(\lambda)q(\lambda)w(\lambda)d\lambda. \tag{5.13}$$

The polynomial s_{k-1} is called the least square polynomial.

The implemented algorithm will now be explained, it is based on the approach taken by [18]. An approximation of the spectrum of A, λ_{min} and λ_{max}, is obtained by using a small number of iterations of the Lanczos method, and then using Lapack's STEQR routine to extract the eigenvalues of the resulting tridiagonal matrix. Lanczos algorithm is an iterative algorithm based on the Krylov subspace for finding eigenvalues and eigenvectors of square matrices. At iteration m the method produces a tridiagonal matrix T being similar to A. Similar means that $T = B^{-1}AB$ for some invertible square matrix B. Similar matrices share the same eigenvalues under certain conditions [37]. The Lanczos algorithm is known to, after few iterations, obtain a good approximation of the extremal eigenvalues, and is thus suitable for our purpose. The STEQR routine calculates the eigenvalues of a tridiagonal matrix [40].

Heuristically, it is often favorable for the convergence of iterative methods to add an extra term to the largest eigenvalue found with Lapack [18]. This increase the probability that the approximated eigenvalues covers the actual spectrum of A. The same term as in [18] was being used, namely the absolute value of the largest eigenvalue's eigenvector's last element. The Lanczos algorithm implemented is detailed in Algorithm 9. To find the approximate eigenvalues, Lapack's STEQR routine was used on the tridiagonal matrix obtained from the Lanczos algorithm.

Algorithm 9 Lanczos algorithm

Choose $v_{:,1}$ so that $\|.\|_2 = 1$ {for example all random}
$\beta_1 = 0$, $V_{:,0} = 0$
for $i = 1...m$ **do**
  $w_i = AV_{:,i} - \beta_i V_{:,i-1}$
  $\alpha_i = (w_i, v_i)$
  $w_{:,i} = w_{:,i} - \alpha_i V_{:,i}$
  if $i < m$ **then**
    $\beta_{j+1} = \|w_i\|_2$
    $V_{:,i+1} = w_j/\beta_{j+1}$
  end if
end for

Ideally we want to find $s(t)$ for which $s(A) = A^{-1}$ acts the same when applied to a vector or matrix. However, it is more efficient to approximate s with a polynomial s_{k-1} of degree less than k. This polynomial should reassemble s as $k \to \infty$. It is chosen to approximate s in a least square sense according to [41]

$$s_{k-1}(t) = \sum_{i=1}^{k-1} (s, P_i) P_i, \tag{5.14}$$

in which $\{P_i\}$ is an orthonormal basis of polynomials in some L_2 space, to be chosen later. For the polynomials a Chebyshev basis was chosen with specific weights to define the inner product in the L_2 space, as was done in [41]. This choice leads to inner products which can be computed explicitly, avoiding numerical integration, therefore reducing computational time. The reason is that Chebyshev polynomials are orthogonal on the interval [−1,1] with respect to the weight $w = \frac{1}{\sqrt{1-x^2}}$. Therefore the interval $[\lambda_1, \lambda_N]$ need to be transformed to [−1,1]. The inner product between Chebyshev polynomials on the interval [−1,1] using the specified weight is

$$(C_i, C_j)_w = \begin{cases} 0, & i \neq j \\ \pi, & i = j = 0 \\ \frac{\pi}{2}, & i = j \neq 0 \end{cases} \tag{5.15}$$

The polynomial basis $\{P_i\}$ can be found using Stieltjes procedure, which satisfies the following 3-term recurrence [41]

$$P_{i+1}(t) = \frac{1}{\beta_{i+1}} (P_i(t)(t - \alpha) - \beta_i P_{i-1}(t)), \;\; j = 1...m. \tag{5.16}$$

In Algorithm 10 a modified Stieltjes procedure used in the implementation is shown. The modification of the procedure is that γ is calculated explicitly, as it will be used in a later step. The starting polynomial S_0 depends on the basis used, and was chosen as $S_0 = t$.

Algorithm 10 The modified Stieltjes procedure used [18].

Initialize $S_0(t)$
$\beta_1 = \|S_0(t)\|$, $P_0 = 0$
$P_1(t) = \frac{1}{\beta_1} S_0(t)$
$\gamma_1 = (s(t), P_1(t))$
for $i = 1...m$ **do**
  $\alpha_i = (tP_i, P_i)$
  $S_i(t) = tP_i - \alpha_i P_i - \beta_i P_{i-1}$
  $\beta_{i+1} = \|S_i\|_2$
  $P_{i+1} = S_j/\beta_{i+1}$
  $\gamma_{i+1} = (s(t), P_{i+1}(t))$
end for

At this point, everything required to set up the polynomial preconditioner has been done. Now the procedure of applying it will be described. Assume that the system to solve is $Ax = b$ and that the initial guess is chosen as x_0, which yields an initial residual as $r_0 = b - Ax_0 = Ax - Ax_0$. Then x can be expressed as

$$x = A^{-1}(r_0 + Ax_0) = x_0 + A^{-1}r_0. \tag{5.17}$$

This in turn can be approximated, using (5.14), as

$$x = x_0 + \sum_{i=1}^{k} \gamma_i P_i(A) r_0 = x_0 + \sum_{i=1}^{k} \gamma_i v_i, \qquad v_i = P_i(A) r_0, \quad \gamma_i = (s,P_i), \tag{5.18}$$

where it becomes apparent why γ was being stored in the Stieltjes procedure. With the help of (5.16), the following recurrence, yielding the v_i's from α and β computed in the Stiletjes procedure, can be derived as

$$v_{i+1} = \frac{1}{\beta}(Av_i - \alpha_i v_i - \beta_i v_{i-1}). \tag{5.19}$$

In the approximation of z from a system $Mz = b$, the final expression in (5.18) combined with the recurrence (5.19), is used to iteratively build the solution. The procedure is described in Algorithm 11. Note that the described operation to apply the preconditioner consists only of vector-vector and SpMV operations, making it suitable for parallel architectures such as GPUs.

Algorithm 11 Applying least square preconditioner

input: x_0, b, α, β
$v_0 = 0$, $v_1 = r_0/\beta$
for $i = 1...m$ **do**
  $x_i = x_{i-1} + \gamma_i v_i$
  $v_{i+1} = \frac{1}{\beta_{i+1}}(Av_i - \alpha_i v_i - \beta_i v_{i-1})$
end for
$x_{m+1} = x_m + \gamma_{m+1} v_{m+1}$

There are two parameters to specify when using LSP preconditioning: the number of iterations in the Lanczos step, and the degree of the preconditioning polynomial. The more Lanczos iteration steps, the tighter bound of the spectrum of A is given, speeding up convergence. The higher degree polynomial, the more exact the preconditioner will be, generally resulting in fewer iterations, but in turn applying the preconditioner will be significantly slower. Caution should be taken when using a polynomial of a high degree, as experiments suggest that numerical errors tend to increase for higher degree polynomials [42]. Usually polynomials of small degree < 15 are most efficient. Advantages of the least square polynomial preconditioner is that a three term recurrence formula can be used for building and applying the preconditioner, and that applying the preconditioner can be efficiently parallelized.

To conclude, three steps need to be implemented to use least square polynomial preconditioning, whose procedure is given in Algorithm 9, 10 and 11 respectively:

- Get an approximate bound of the spectrum of A, preferably using a few steps of Lanczos method, Algorithm 9, combined with Lapacks STEQR routine.
- Calculate the coefficients α, β, and the product γ using Stieltjes procedure, Algorithm 10, preferably using Chebyshev polynomials transformed to the interval $[-1, 1]$.
- Implement the routine for carrying out the preconditioning operation, Algorithm 11, i.e. approximating z from $Mz = b$.

5.3 Other methods

In addition to the Krylov subspace iterative methods and preconditioners there exist two other popular techniques for solving sparse linear systems. The first one is to use the Fast Fourier Transform (FFT). This technique is hard to use for complex geometries and boundaries and will not be further discussed in this thesis. The other one is multigrid methods. These methods can be highly efficient if used properly. Multigrid methods are often used for solving the Poisson pressure equation derived when solving the Navier-Stokes equations. It can be implemented to be optimal in complexity for solving sparse linear systems, i.e. with complexity $\mathcal{O}(n)$ [35].

Recall that the iterative methods tend to smoothen high frequency errors rather quick while taking a much longer time, $\propto (h^{-1})$, to get rid of low frequency errors. To overcome this, multigrid methods have been developed. In the methods, the full grid of the solution domain is transformed to coarser grids recursively in different levels. The problem on the coarser grids are solved and smoothed using direct methods and iterative solvers, since low frequency errors in the finer problem will appear as high frequency errors on the coarser grids. The procedure could be as follows, but varieties exist: i) Perform a few steps of an iterative method. ii) Project the grid into a coarser grid with a subset of the original grid points, recursively go back to step i) until the grid is coarse enough. iii) When going back up the recursion tree, interpolate the solution obtained for the coarse

grid points to the finer grid and perform a few steps of an iterative method to smooth out the errors.

There exist two varieties of multigrid methods, Geometric MultiGrid (GMG) and Algebraic MultiGrid (AMG) methods. In GMG methods, the starting point is the coarse geometrical mesh, subsequently refining the grid. While in AMG methods the starting point is the algebraic system of equations, coarsening the grid on purely algebraic foundations. AMG can thus be used as a black box solver. Multigrid techniques exist both as a solver and as a preconditioner.

The difficulties with multigrid lies in being able to handle complex domains efficiently and that they have many dependent parameters. Implementing multigrid methods are not in the scope of this thesis. Due to its difficulty in constructing a general solver and the complexity. However, the implemented Navier-Stokes solver was compared to the best available CPU implementation, which uses AMG to solve the pressure Poisson system of equations.

5.4 Implementation considerations of iterative solvers and preconditioners

The iterative solvers were implemented to a large extent using elementary operations from existing software libraries. Most notably Nvidia's GPU linear algebra libraries cuSPARSE and cuBLAS, but also a Lapack [40] routine were used. Several functions for combined operations were implemented in CUDA as part of this thesis work. This was the case when no efficient implementation was available for the desired operation or when there existed a potential performance increase when combining several operations to one. The algorithms are built up using elementary operations of the discussed libraries as building blocks, combined to an algorithm in this thesis work. The motivation of using existing routines to a high extent was that of productivity and increased performance, due to highly optimized code by the Nvidia developers. The Nvidia libraries were faster than the open source alternatives for the tested libraries, as could be seen inTable 4.1 and 4.2.

The operations implemented in CuSPARSE and CuBLAS are a subset of the BLAS elementary linear algebra routines such as SpMV, AXPY, dot nrm2 etc. The only iterative method or preconditioner method used in this thesis not implemented by the author is the Incomplete LU factorization, instead Nvidia's implementation in cuSPARSE was used. However as it turns out, the LU factorization was only evaluated for performance and not actually used in the flow solver implementation as other methods were faster.

The main CUDA optimization consideration for the implementation of linear algebra routines was using vectors and matrices aligned in appropriate order, and ensuring memory accesses were aligned in the implemented functions.

5.5 Results

The implemented iterative solvers and preconditioners were validated and tested for various aspects, such as comparison with CPU and other GPU accelerated linear algebra packages. They were validated on different matrices relevant to CFD and the discretization used. The hardware used to test the solvers were an Intel Xeon E5-2650 CPU and an Nvidia Geforce GTX 660 Ti GPU, see Table 2.1 for details. A relative residual of 10^{-9} was set as convergence criterion, which is a fairly tough restriction.

The CPU solvers tested against were solvers from Intel's Math Kernel Library, MKL [43]. MKL includes the restarted GMRES method, the CG method as well as ILU0 and Jacobi preconditioners among others. The implementation is threaded to use all CPU cores available. The MKL library is a standard choice for many applications and highly optimized.

The implemented solvers and preconditioners were tested on two different external GPU libraries to ensure their implementation efficiency. One of the libraries was CUDA ITSOL [33], a CUDA accelerated linear algebra and sparse solver library developed by Ruipeng Li under supervision of Yousef Saad. The other was Paralution-0.1.0 [34] a multi/many core CPU/GPU powered sparse solver library developed by Dimitar Lukarski from Uppsala University.

5.5.1 Performance of the implemented solvers

The implemented solvers and preconditioners, BiCGStab, GMRES, CG, diagonal scaling and least square polynomials along with Nvidia's implementation of an ILU0 preconditioner were tested on a set of four test matrices. The test matrices were chosen with respect to the purpose of this thesis, namely CFD matrices resulting from FVM discretization of the Navier-Stokes equations: uniform grid momentum matrix and pressure matrix, and refined grid momentum matrix and pressure matrix were tested. The matrices as well as their characteristics are tabulated in Table 5.1.

Table 5.1: Properties of the validation matrices.

Matrix	**Rows**	**nz/row**	**Properties**
momentum1M	1 000 000	6.94	-
momentumR1.7M	1 760 792	7.12	-
pressure1M	1 000 000	6.94	sym. pos def
pressureR1.7M	1760792	7.12	sym. pos def

5.5.2 Properties and timings of the implemented solvers

To get an idea of the time consuming parts of the iterative solvers, the time distribution per iteration of the algorithms was evaluated. The results can be seen in Figure 5.1. CG requires least time per iteration, about half that of BiCGStab, while GMRES requires

most. Note that the time per iteration shown in the figure for GMRES is an average of the time per iteration for a complete restart cycle. Earlier iterations of the cycle, when the Krylov subspace stored is small, are less time consuming than later iterations. The large time contribution from BLAS-1 routines in GMRES comes from the loop required to build the Hessenberg matrix. GMRES is also the only of the three methods where some non-negligible operations are performed on the host, which can be seen by the "Misc" column in the figure. This also tells us that the total time required for CPU routines is relatively small. The time distribution plot reveals that a performance improvement of SpMV would achieve a significant speedup. It also tells us that optimizing the BLAS1 operations might be worth considering.

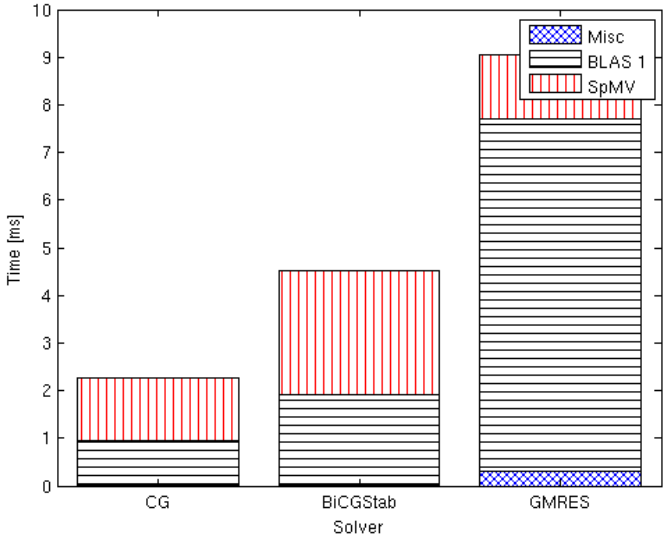


Figure 5.1: Time per iteration without preconditioner for the implemented iterative solvers, evaluated on the pressureR1.7M matrix. For GMRES, the timing is the average time required per iteration for a Krylov subspace of size 40.

In Figure 5.2 the computational time for several of the implemented solver and preconditioner configurations for the four test matrices from Table 5.1, can be seen. Note that a logarithm scale for time on the y-axis have been used. The timings vary considerably for different configurations. Worth to notice is that the Jacobi preconditioned methods were best in the case when an efficient solver for the specified matrix has been chosen. BiCGStab was best for the momentum matrices while CG was most efficient for the pressure matrices. As a slight surprise it seems that the tradeoff in extra computational time versus decrease in condition number/number of iterations required for convergence when using an advanced preconditioner, does not seem to payoff for the tested matrices. The exception is for some cases when the matrix is ill-conditioned or not well suited for the specific iterative method. An example being ILU0 preconditioned GMRES against Jacobi preconditioned GMRES for the pressure matrices, which may imply that for

even larger cases advanced preconditioners can perform better. Specifically the least square polynomial preconditioner applied to CG appears to perform worse than Jacobi preconditioned CG. In the case of the ILU0 preconditioner, one can think that the difficulty in effectively parallelizing it makes a couple of extra iterations more efficient than using the preconditioner. The GMRES solver, however, seems to benefit more from preconditioning than CG and BiCGStab. One should also note that the least square polynomial might be an alternative preconditioner even for the momentum matrices, even though it is only guaranteed to reduce the condition number if the matrix is symmetric.

A scaling study was made on a uniform grid to see how the GPU solvers scale to larger grids. The results can be seen in Figure 5.3, where nominal times and iterations are plotted versus the size of the matrix. The matrix is taken from a 3D uniform pipe flow CFD problem with a sphere located in the middle. The number of iterations scales linearly to the number of nodes, which is in accordance with the theory. The solution times, being a product of a linear increase in the number of iterations and a linear increase in the amount of work performed per iteration, increase super linearly.

It should also be noted that the number of iterations of the momentum matrices are significantly fewer than those required for the pressure matrices, thus indicating that the setup of the preconditioner are not as efficient on the GPU as on the CPU.

5.5.3 Performance comparison to a CPU library and other GPU libraries

Speedup of the implemented GPU solvers compared to CPU equivalents of the same iterative method and preconditioner configuration can be seen in Figure 5.4. The equivalent solvers are taken from Intel's MKL library. The implemented GPU solvers without preconditioners was around seven times faster for the iterative methods. When the ILU0 preconditioner was used the speedup was only around a factor two to four, indicating that the implementation of ILU0 on the GPU does not gain as high speedup as the iterative methods alone.

In order to ensure the effectiveness of the implemented solvers, they were tested against the CUDA ITSOL and Paralution iterative solvers open source libraries. In the comparison, the most efficient solver and preconditoner configuration found for each test matrix and solver library was chosen. Often the optimal choice of solver was the same for the different solver libraries. In Figure 5.5, the speedup of the implemented GPU solvers against CUDA ITSOL and Paralution can be seen. Note that the measured times are including the setup of a preconditioner (if applicable). Therefore, a fast solver which require a long time for preconditioner setup time, might be more efficient if systems with the same matrix is solved for several right-hand-sides. The solvers implemented in this thesis was faster than the open source libraries. Ensuring the efficiency of chosen methods, algorithms and implementations. The advantage of the implemented solvers over the open source libraries can find some explanation in the following. The BLAS kernels in cuSPARSE and cuBLAS might be better optimized for the specific GPU tested. The other libraries are more general, which requires extra overhead. Observe further that even though the testing of different solvers and preconditioner combinations were

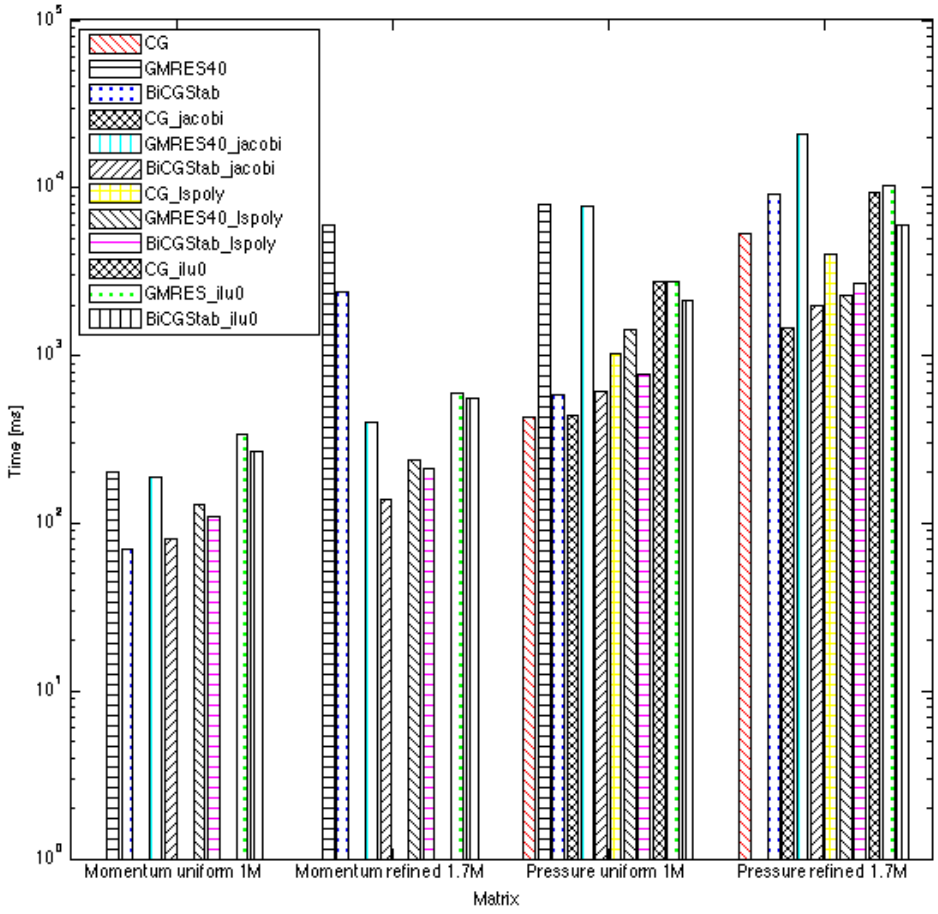
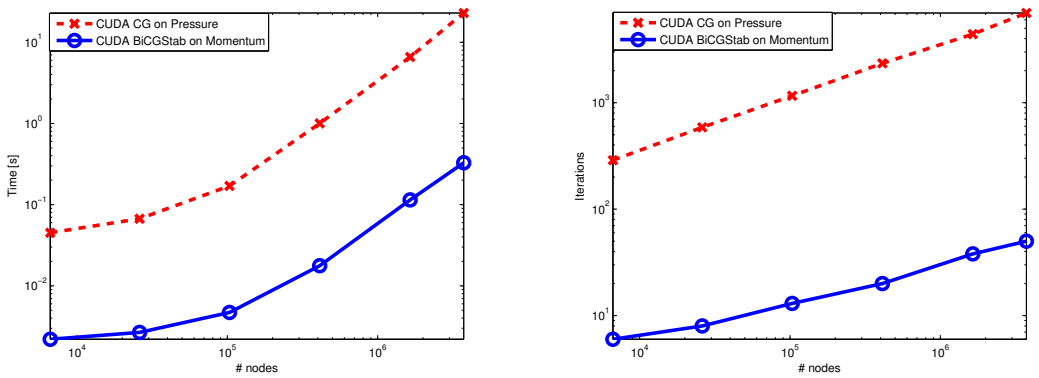


Figure 5.2: Comparison of solution times for individual matrices for the implemented iterative methods and preconditioners. The staples are in the same order as the legend says. When there is an empty staple it means that the solver was not used due to wrong matrix type or it did not converge.



(a) Time in seconds. **(b)** Number of iterations.

Figure 5.3: Scaling of solution times and number of iterations required for CG and BiCGStab methods on the momentum and pressure matrices, respectively, originating from a uniform grid 3D pipe flow problem around a sphere.

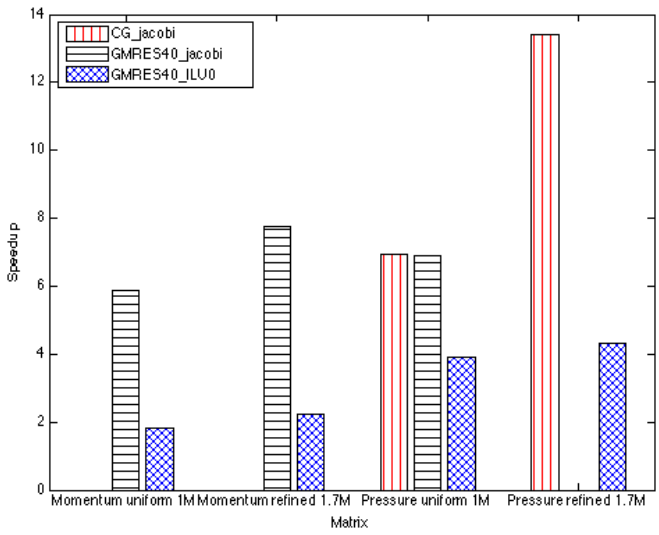


Figure 5.4: Speedup of the GPU solvers compared to CPU implementations of the same solver and preconditioner configurations in Intel's MKL library.

extensive, there is no absolute guarantee that the optimal combination was found for all tested libraries.

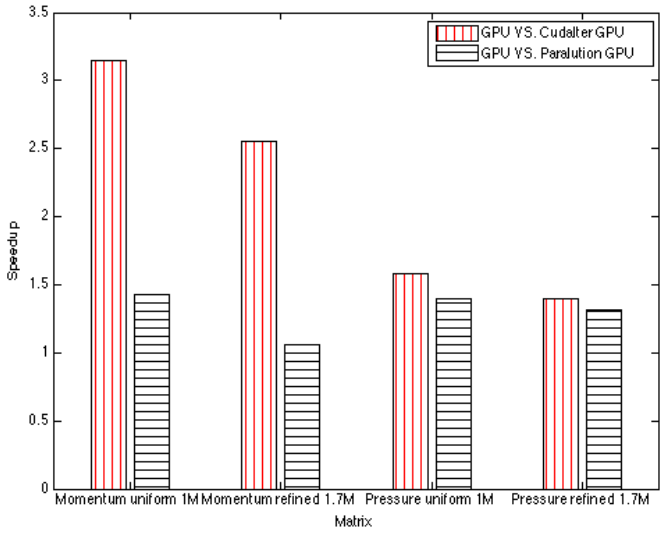


Figure 5.5: Speedup of the implemented GPU solvers compared to the CUDA Itsol and Paralution-0.1.0 open source libraries. The iterative method and preconditioner found most efficient for each library and test matrix is shown.

Chapter 6

The Navier-Stokes equations solver

The Navier-Stokes equations describes the motion of fluids. Its application in physics and engineering applications are many. It is used in the modeling of weather systems, blood flow, cooling systems for computers, design of cars, boats, and airplanes, to name a few. Mathematically, Navier-Stokes equations and simplifications of it usually belongs to a classification of PDEs called parabolic or hyperbolic, depending on coefficients and what simplifications are made [44], but with particular simplifications it can also be elliptic.

For many applications, simplifications to the general Navier-Stokes equations can be made. In this thesis, a solver for the incompressible Navier-Stokes equations is implemented. In incompressible flow, the density is assumed to be constant, $\frac{\partial \rho}{\partial t} = 0$, which is equivalent to a divergence free velocity field. It is generally a valid assumption for fluid flow at low speeds, more specifically for Mach numbers $Ma = \frac{v}{v_{sound}} < 0.3$. Following is the incompressible Navier-Stokes equations in conservative form

$$\rho(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u}) = -\nabla p + \mu \Delta \mathbf{u} + \mathbf{f}, \tag{6.1}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{6.2}$$

where μ is the dynamic viscosity, ρ the viscosity, $\mathbf{u}$ velocity, p pressure, and $\mathbf{f} = \mathbf{f_{gravity}} + \mathbf{f_{drag}} + \mathbf{f_{natural}}$ is the force term accounting for external forces. The force term is sometimes also called the source term. The first equation (6.1) is called the momentum equation. It describes conservation of momentum of fluid flow. The second equation (6.2) is called the continuity equation. It describes the conservation of mass. The first term on the left hand side of the momentum equation is called the transient term and the second the convective term. On the right hand side of the momentum equation, the first term describes forces from a change in pressure and the second the diffusion. Notice that the momentum equation is a vector relation and can in 3D be split up into three scalar equations, one for each velocity component u, v and w. The four unknowns of the Navier-Stokes equations are u, v, w, and p.

A general procedure for solving the Navier-Stokes equations is to, in each time step,

converge towards the solution via an iterative procedure. The implementation in this report is essentially: 1) Make a guess of the pressure. 2) Solve **u** from (6.1). 3) Correct the pressure and velocity such that (6.2) is exactly satisfied. 4) Go back to step 2 until convergence. The specific method used is the SIMPLEC coupling scheme, it will be described in detail in later sections. Let us now discretize the terms in the momentum equation (6.1), first individually and then obtaining the full discretization by summing them together.

The domain discretization, discretization schemes and boundary treatment adopted in this thesis is inspired by the work [45], even though there are significant differences. Furthermore parts of the high level code structure is inspired by FCC's IBOFlow. IBOFlow has also been used to validate the code during development.

The outline of this chapter will be as follows. First, the momentum equations will be discretized, describing the schemes used and difficulties. Then, the velocity-pressure coupling scheme and discretization of the resulting pressure equation will be detailed followed by a brief motivation and discussion of the methods used. Next, boundary treatment will be explained. General and CUDA implementation considerations is described after. Finally, validation and performance results of the implemented solvers, and comparisons to CPU and a commercial solver will be presented.

6.1 Discretization

There exist several general methods to discretize the Navier-Stokes equations (6.1) and (6.2. In the Lagrangian formalism, the fluid is treated as particles while in the Eulerian formalism the rate of change in discretized volumes, often called Control Volumes (CV), is studied. The Eulerian approach is most common and also the one chosen in this thesis. Two common choices using the Eulerian formulation are the Finite Element Method (FEM) and the Finite Volume Method (FVM). FVM is a mix between FEM and Finite Difference Methods, taking advantage of FEM's geometrical flexibility while using FDM's discretization of variables. FVM is employed in this thesis due to its relative simplicity over FEM, speed considerations and intuitive modeling. Its drawback is that it lacks higher order accuracy due to difficulties in exact approximations of derivatives, while in FEM on the other hand more accurate basis functions can be used and the use of a weak formulation can transform higher order derivatives to lower ones. However the accuracy of FVM is sufficient for the purposes of this thesis.

6.1.1 The finite volume discretization

In the Finite Volume Method the domain of interest is split up into small control volumes, sometimes denoted cells. The idea is to ensure that the conservation equations are satisfied in control volume. This is done by integrating (6.1) over each control volume and solving the resulting system of equations. In this thesis, the control volumes were chosen as squares in a Cartesian grid, see Figure 6.1.

In general two techniques are used when assigning pressure and velocity values to

nodes. In a collocated grid arrangement, pressure and velocity are assigned in the same nodes, in a staggered grid the velocities are defined on the faces of the pressure cells and vice versa. Staggered grids have an advantage over collocated arrangements originating from numerical issues in the discretization. If one were to do a simple linear interpolation of ∇p in a cell with a colocated grid, the resulting expression is independent of the pressure in the actual cell itself. This can lead to oscillations of the pressure field in the solution, a so-called checkerboard pressure. Checkerboard pressures do not arise on staggered grids. There exist techniques to prevent pressure oscillations also in collocated grid arrangements, described in detail later on. In this thesis, a collocated grid setup is being used.

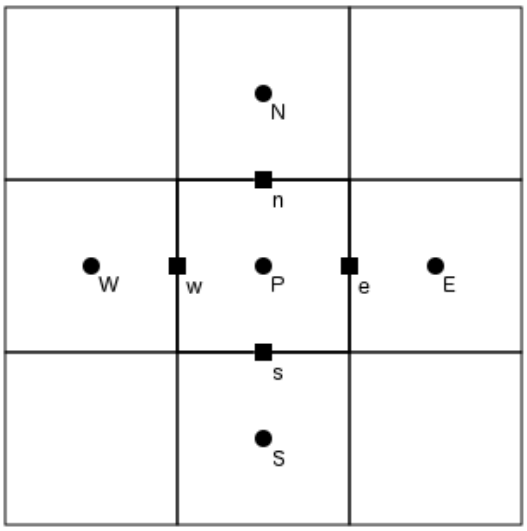


Figure 6.1: The collocated grid arrangement used, projected onto 2D. Each square represents a cell in the finite volume discretization. Properties are stored at the cell centers, denoted by capital letters. Properties required on cell faces are interpolated from the stored ones, denoted by lower case letters.

6.1.2 Discretization of the transient term

In this and the following sections, the different terms of the 3D momentum equation will be discretized in the x direction over a control volume. The y and z directions follow analogously. Each term is integrated over the control volume, see Figure 6.1, and a time window. The expression for the transient term becomes

$$\int_t^{t+\Delta t}\int_b^t\int_s^n\int_w^e \rho\frac{\partial u}{\partial t}dxdydzdt = \rho_p V[u]_t^{t+\Delta t} = \rho_p V(U_p - U_p^0). \tag{6.3}$$

U_p^0 is the velocity from of previous time step and U_p the velocity of the next time step, i.e. the velocity sought for. The density, ρ, has been approximated to be constant over the

control volume with the value chosen at the cell center. The integration over a dimension of the volume is approximated by the value in the midpoint times the integration length. This is known as the midpoint rule or rectangle rule. For more on quadrature rules consult [46] etc. For a quantity $\Phi(x)$ then,

$$\int_w^e \Phi(x)dx = \Phi\left(\frac{e+w}{2}\right)(e-w) + \mathcal{O}((e-w))^2, \tag{6.4}$$

putting it in our grid notation and throwing away higher order terms we get

$$\int_w^e \Phi(x)dx = \Phi_P \Delta x. \tag{6.5}$$

The midpoint quadrature rule is second order accurate, it will be used without further notice wherever applicable for the rest of the derivations in this chapter.

6.1.3 Discretization of the pressure term

The pressure term describes pressure forces in the momentum equation. Integrating the pressure term over the control volume and over a time window results in

$$\begin{aligned}
&\int_t^{t+\Delta t}\int_b^t\int_s^n\int_w^e -\frac{\partial p}{\partial x}dxdydzdt = -\Delta t\Delta y\Delta z[p]_e^w = \\
&-\Delta t\Delta y\Delta z\left(\frac{P_E+P_P}{2} - \frac{P_P-P_W}{2}\right) = \\
&-\Delta t\Delta y\Delta z\left(\frac{P_E+P_W}{2}\right) = -\Delta t DP_P, \quad DP_P = \Delta y\Delta z\left(\frac{P_E+P_W}{2}\right).
\end{aligned} \tag{6.6}$$

Simple linear interpolation have been adopted to approximate the pressure at the faces. Note that the expression is independent of the pressure in the actual cell. A problem we will come back to, and find a solution to, when coupling the momentum equations with the pressure equation.

6.1.4 Discretization of the source term

The source term describes the effect of external forces on the flow. General forces are buoyancy forces coming from the gravity, drag forces if there exist an interface between different flows (multi phase flows), and other natural forces. It is integrated over a control volume and time window

$$\int_t^{t+\Delta t}\int_b^t\int_s^n\int_w^e S dxdydzdt = \overline{S_p}\Delta x\Delta y\Delta z\Delta t, \tag{6.7}$$

where $\overline{S}$ denotes the interpolation of S to the current node.

6.1.5 Discretization of the diffusive term

The diffusive term describes the diffusion of momentum as a consequence of viscous forces. Integrating it over a control volume and over a time window yields

$$\begin{aligned}&\int_t^{t+\Delta t}\int_b^t\int_s^n\int_w^e\left(\frac{\partial}{\partial x}\left(\mu\frac{\partial u}{\partial x}\right)+\frac{\partial}{\partial y}\left(\mu\frac{\partial u}{\partial y}\right)+\frac{\partial}{\partial z}\left(\mu\frac{\partial u}{\partial z}\right)\right)dxdydzdt=\\&\Delta t\left(\Delta y\Delta z\left(\mu\frac{\partial u}{\partial x}\right)_w^e+\Delta x\Delta z\left[\mu\frac{\partial u}{\partial y}\right]_s^n+\Delta x\Delta y\left[\mu\frac{\partial u}{\partial z}\right]_b^t\right).\end{aligned}\tag{6.8}$$

Using central differencing to evaluate the derivatives on the faces, giving for the x derivative $\frac{\partial u_e}{\partial x}=\frac{U_E-U_P}{\delta x}$, yields

$$\begin{aligned}&\Delta t(\mu_e\frac{U_E-U_P}{\delta x}\Delta y\Delta z-\mu_w\frac{U_P-U_W}{\delta x}\Delta y\Delta z+\\&\mu_n\frac{U_N-U_P}{\delta y}\Delta x\Delta z-\mu_s\frac{U_P-U_S}{\delta y}\Delta x\Delta z+\\&\mu_t\frac{U_T-U_P}{\delta z}\Delta x\Delta y-\mu_b\frac{U_P-U_B}{\delta z}\Delta x\Delta y),\end{aligned}\tag{6.9}$$

Introducing the coefficients

$$\begin{aligned}&a_{E,d}=\mu_e\frac{\Delta y\Delta z}{\delta x},\quad a_{W,d}=\mu_w\frac{\Delta y\Delta z}{\delta x},\quad a_{N,d}=\mu_n\frac{\Delta x\Delta z}{\delta y},\quad a_{S,d}=\mu_s\frac{\Delta y\Delta z}{\delta y}\\&a_{T,d}=\mu_t\frac{\Delta x\Delta y}{\delta z},\quad a_{B,d}=\mu_b\frac{\Delta x\Delta y}{\delta z},\quad a_{P,d}=\sum_{NB=\{E,W,N,S,T,B\}}a_{NB,d},\end{aligned}\tag{6.10}$$

reduces the expression (6.9) to

$$\Delta t\left(\sum_{NB=\{E,W,N,S,T,B\}}a_{NB,d}U_{NB}-a_{P,d}U_P\right).\tag{6.11}$$

In the simplified case of uniform grid of cell side length, L, with constant viscosity, μ, the coefficients in (6.10) are reduced to

$$a_{E,d}=a_{W,d}=a_{N,d}=a_{S,d}=a_{T,d}=a_{B,d}=\mu L.\tag{6.12}$$

An important detail which has been swept under the carpet in the discretization, is how the quantities u, p, and S should be interpolated in the temporal dimension. Recall that an integral over a time window, $[t,t+\Delta t]$, was introduced. This requires a way to approximate U during this time interval. A simple linear interpolation in time would yield

$$U=cU^{t+\Delta t}+(1-c)U^t,\tag{6.13}$$

with c chosen as 0.5. This particular time discretization scheme is called a Crank-Nicholson scheme. It is second order accurate in time. Choosing $c=0$ gives a fully

explicit scheme $U = U^t$, which is first order accurate and often unstable. Choosing $c = 1$ yields a fully implicit scheme, which is first order accurate in time and unconditionally stable. Unconditionally stable means that large time steps can be taken without loss in stability, but large time steps still give poor accuracy. One can also think of higher order schemes. In this derivation and the following, the fully implicit scheme will be used. This is partly because of its relative simplicity of expressions, but also because of its robustness and common use in CFD codes.

6.1.6 Discretization of the convective term

The convective term accounts for the presence of convective acceleration, which is an effect of spatially varying velocity. Recall from (6.1) that the convective term is non linear, being a product of two linear terms. For this reason the term is linearized in order to be able to solve it from a linear system of equations. It is linearized by assuming that the first factor in the product $u\frac{\partial u}{\partial x}$ is known, let us denote this quantity by a hat, $\hat{u}$. Integrating the x component of the convective term over the control volume and over a time window using this assumption yields

$$\begin{aligned}
&\int_t^{t+\Delta t}\int_b^t\int_s^n\int_w^e \rho(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial v} + w\frac{\partial u}{\partial z}) = \\
&\Delta t\rho([\hat{u}u]_w^e\Delta y\Delta z + [\hat{v}u]_n^s\Delta x\Delta z + [\hat{w}u]_t^b\Delta x\Delta y) = \\
&\Delta t\rho\{\ [(\hat{U}U)_e - (\hat{U}U)_w]\Delta y\Delta z + [(\hat{V}U)_n - (\hat{V}U)_s]\Delta x\Delta z + \\
&[(\hat{W}U)_t - (\hat{W}U)_b)]\Delta x\Delta y\},
\end{aligned} \tag{6.14}$$

where a fully implicit time discretization scheme have been used for the velocities U, V and W. The explicit velocities $\hat{U}$, $\hat{V}$ and $\hat{W}$ are taken to be those of the previous iteration step. These are, for now, interpolated by simple linear interpolation such that $\hat{U}_e = \frac{\hat{U}_E + \hat{U}_W}{2}$.

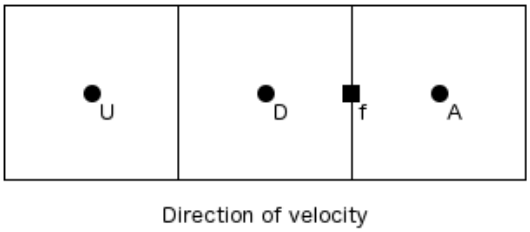


Figure 6.2: The velocity interpolation stencil. Assuming the flow is from left to right, U, D, and A denotes the upwind, donor and acceptor cell respectively.

The implicit velocities U, V and W are interpolated to the faces by a convective scheme, which is just a family of interpolation schemes. Suppose the flow is in the direction shown in Figure 6.2. With the same notation the face velocity can now be interpolated as

$$U_f = c_A U_A + c_D U_D + c_U U_U, \tag{6.15}$$

where the subscript A denotes the acceptor cell, D the donor cell and U the upwind cell. A common and very simple scheme is the upwind scheme, where $c_A = 1, c_D = c_U = 0$, similarly the downwind scheme is given by $c_D = 1, c_A = c_U = 0$ and the central scheme by $c_A = 0.5, c_D = 0.5, c_U = 0$. The first and second are first order accurate while the third scheme is second order accurate.

Using the first order upwind scheme for interpolation of the implicit face velocities, assuming the flow is in positive x, y and z directions respectively, and linear interpolation for the explicit "hat" velocities the discretization of the convective term becomes

$$\begin{aligned}\Delta t\rho \Bigg(&\left(\frac{\hat{U}_E + \hat{U}_P}{2} U_E - \frac{\hat{U}_P + \hat{U}_W}{2} U_P \right) \Delta y \Delta z + \\ &\left(\frac{\hat{V}_N + \hat{V}_P}{2} U_N - \frac{\hat{V}_P + \hat{V}_S}{2} U_P \right) \Delta x \Delta z + \\ &\left(\frac{\hat{W}_T + \hat{W}_P}{2} U_T - \frac{\hat{W}_P + \hat{W}_B}{2} U_P \right) \Delta x \Delta y \Bigg).\end{aligned} \tag{6.16}$$

6.1.7 Resulting discretization of the full momentum equations

Let us now add the discretization of all terms, group together terms and divide by Δt. Then the full discretized form appears as

$$a_P U_P - \sum_{NB=E,W,N,S,T,B} a_{NB} U_{NB} = \Delta V \overline{S} - DP_P + a_P^0 U^0, \tag{6.17}$$

where $a_P = a_P^0 + a_{P,d} + a_{P,c}$, $a_P^0 = \frac{\rho \Delta V}{\Delta t}$ and $a_{NB} = a_{NB,d} + a_{NB,c}$. $a_{.,c}$ and $a_{.,d}$ are the coefficients for a velocity at a grid point coming from discretization of the convective and diffusive terms respectively. This is a system of equations where dependent variables are on the LHS and the explicit terms are on the RHS for each control volume. For the whole domain the resulting system is of the form $AU = b$ where the velocity in the center of each cell is the value of U in that cell. Analogously systems for the V and W velocities can be derived.

6.1.8 Pressure and velocity correction equations

Recall that for the expressions of the momentum equations, a guessed pressure field was used which might be far from correct. This results in an error in the continuity equation, which leads to a non-divergence free flow. To correct the velocity and pressure, the continuity equation is used. In this thesis, a variation of coupling algorithm originating from the SIMPLE, Semi-Implicit Method for Pressure Linked Equations, algorithm [47] called SIMPLEC, SIMPLE-Consistent, is being used. In the scheme, the continuity equation is exactly satisfied after the correction step. Other correction schemes exist.

Originating from the SIMPLE family are SIMPLER (SIMPLE-Revised), and PISO (Pressure Implicit with Splitting Operator). Another method sometimes used is a projection method called Fractional Step Method. The pros and cons of different schemes will be discussed in the next section. For a complete description of their implementation the reader is referred to educational textbooks such as [44].

The correction introduced to satisfy the continuity equation will make the momentum equation be dissatisfied. The SIMPLEC algorithm is therefore an iterative procedure, converging towards the solution. The resulting correction equation will be a Poisson equation for pressure. The continuity equation is a first order spatial derivative of velocity, while the velocity used contains a first order spatial derivative of pressure, together with known terms.

Denote the approximate velocities obtained when solving the momentum equations by u^*,v^*, and w^* and the guessed pressure used by p^*. The correct velocities can be written as

$$\begin{aligned} u &= u^* + u', \\ v &= v^* + v', \\ w &= w^* + w', \\ p &= p^* + p', \end{aligned} \tag{6.18}$$

where the prime terms are the correction terms. Subtracting the starred momentum equation, u^*, from the actual momentum equation u, i.e. u^* and u in (6.17), results in

$$a_P(U - U^*) - \sum_{NB=E,W,N,S,T,B} a_{NB}(U_{NB} - U^*_{NB}) = -(DP_P - DP^*_P), \tag{6.19}$$

where we recall that $DP_P = \frac{P_E + P_W}{2}$. Substituting the expressions for the corrections (6.18) into the above equation yields

$$a_P U' - \sum_{NB=E,W,N,S,T,B} a_{NB} U'_{NB} = -DP'_P. \tag{6.20}$$

If we at this point neglect the neighboring velocities in (6.20), we end up with the expression used in the SIMPLE algorithm. However, a less severe approximation would be to approximate the neighboring velocity in (6.20) with the velocity at the current node, $U'_{NB} = U_P \; \forall NB$. Doing this yields the expression for the SIMPLEC algorithm,

$$U' = -\frac{DP'_P}{a_P - \sum_{NB=E,W,N,S,T,B} a_{NB}}. \tag{6.21}$$

The update formula for the U velocity component now becomes

$$U = U^* - \frac{DP'_P}{a_P - \sum_{NB=E,W,N,S,T,B} a_{NB}} = U^* - \frac{DP'_P}{a_{P,Sum}}, \tag{6.22}$$

where the coefficient $a_{P,Sum} = a_P - \sum_{NB=E,W,N,S,T,B} a_{NB}$ have been introduced. The equations for the y and z directions are derived analogously with expressions being

$$V = V^* - \frac{DP'_P}{a_{P,Sum}}, \tag{6.23}$$

$$W = W^* - \frac{DP'_P}{a_{P,Sum}}. \tag{6.24}$$

Note that the velocity corrections are dependent on the pressure correction p' through DP_P.

Now the equations for the pressure correction equation will be derived. Starting from the continuity equation (6.2), integrating it over a control volume and time step, yields

$$\begin{aligned} &\int_t^{t+\Delta t} \int_b^t \int_s^n \int_w^e \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \right) dxdydzdt = 0 \Leftrightarrow \\ &(u_e - u_w)\Delta y \Delta z + (v_n - v_s)\Delta x \Delta z + (w_t - w_b)\Delta x \Delta y = 0. \end{aligned} \tag{6.25}$$

Assuming $\Delta x = \Delta y = \Delta z$, the area expressions in (6.25) can be substituted by an area A. At this point the velocities need to be coupled with the velocities derived from the momentum equation. Recall that in the collocated grid arrangement pressure oscillations might occur if linear interpolation is adopted. The issue is solved by adopting Rhie-Chow interpolation. However, due to simplicity of expressions we will stick to linear interpolation for now and describe the Rhie-Chow interpolation afterwards. Inserting the expressions for the velocity corrections (6.22), (6.22), (6.23), and (6.24) in (6.25), adoptting linear interpolation and using $DP_P = \frac{P_E + P_W}{2} A$ results in

$$\begin{aligned} &\left[\left(\frac{U^*_P + U^*_E}{2} - (P'_E - P'_P) A \frac{1}{2} \left(\frac{1}{a_{E,Sum}} + \frac{1}{a_{P,Sum}} \right) \right) - \right. \\ &\left. \left(\frac{U^*_W + U^*_P}{2} - (P'_P - P'_W) A \frac{1}{2} \left(\frac{1}{a_{P,Sum}} + \frac{1}{a_{W,Sum}} \right) \right) \right] A + \\ &\left[\left(\frac{V^*_P + U^*_N}{2} - (P'_N - P'_P) A \frac{1}{2} \left(\frac{1}{a_{N,Sum}} + \frac{1}{a_{P,Sum}} \right) \right) - \right. \\ &\left. \left(\frac{V^*_S + V^*_P}{2} - (P'_P - P'_S) A \frac{1}{2} \left(\frac{1}{a_{P,Sum}} + \frac{1}{a_{S,Sum}} \right) \right) \right] A + \\ &\left[\left(\frac{W^*_P + W^*_T}{2} - (P'_T - P'_P) A \frac{1}{2} \left(\frac{1}{a_{T,Sum}} + \frac{1}{a_{P,Sum}} \right) \right) - \right. \\ &\left. \left(\frac{W^*_B + W^*_P}{2} - (P'_P - P'_B) A \frac{1}{2} \left(\frac{1}{a_{P,Sum}} + \frac{1}{a_{B,Sum}} \right) \right) \right] A = 0. \end{aligned} \tag{6.26}$$

The expression above can be rearranged to get the final expression for the pressure correction

$$a'_P P'_P - \sum_{NB=E,W,N,S,T,B} a'_{NB} P'_{NB} = b', \tag{6.27}$$

identifying the coefficients as

$$
\begin{aligned}
a'_E &= \left(\frac{1}{a_{P,sum}} + \frac{1}{a_{E,sum}}\right)\frac{1}{2}A^2, \qquad a'_W = \left(\frac{1}{a_{P,sum}} + \frac{1}{a_{W,sum}}\right)\frac{1}{2}A^2, \\
a'_N &= \left(\frac{1}{a_{P,sum}} + \frac{1}{a_{N,sum}}\right)\frac{1}{2}A^2, \qquad a'_S = \left(\frac{1}{a_{P,sum}} + \frac{1}{a_{S,sum}}\right)\frac{1}{2}A^2, \\
a'_T &= \left(\frac{1}{a_{P,sum}} + \frac{1}{a_{T,sum}}\right)\frac{1}{2}A^2, \qquad a'_B = \left(\frac{1}{a_{P,sum}} + \frac{1}{a_{B,sum}}\right)\frac{1}{2}A^2, \\
a'_P &= \sum_{NB=E,W,N,S,T,B} a'_{NB} \\
b' &= \frac{1}{2}\left[(U^*_E - U^*_W)A + (V^*_N - V^*_S)A + (W^*_T - W^*_B)A\right].
\end{aligned}
\tag{6.28}
$$

The only thing dependent on how the face velocities are interpolated is b'. The general expression, using unspecified face interpolation, would be (6.28) but with

$$
b' = \left[(u^*_e - u^*_w)A + (v^*_n - v^*_s)A + (w^*_t - w^*_b)A\right]. \tag{6.29}
$$

To finalize the discretization scheme, the face velocity interpolation for the pressure correction equation will now be addressed. Having a collocated grid makes the velocity at the cell center independent of the pressure at its center. This can give rise to an oscillating pressure field, often referred to as a checkerboard pressure, see [44] for further reading. As proposed by Rhie and Chow in their paper [48] this can be remedied by using a wisely chosen interpolation of the face velocities for the pressure correction, called Rhie-Chow interpolation. The method is often used together with collocated grid arrangements. For the east cell, the expression used for interpolating the u face velocity is given by

$$
u_e = \frac{U_P + U_E}{2} - \frac{1}{2}\left(\frac{1}{a_{P,Sum}} + \frac{1}{a_{E,Sum}}\right)(P_E - P_P) + \frac{1}{2}\left(\frac{DP_P}{a_{P,Sum}} + \frac{DP_E}{a_{E,Sum}}\right), \tag{6.30}
$$

the other face velocities are given analogously.

By expanding the pressure dependent terms of the Rhie-Chow interpolation and comparing it to the third derivative of pressure across a face, it can be proven that these are equal up to third order. Effectively meaning that Rhie-Chow interpolation adds a third derivative of pressure to the linear interpolation of velocity, thereby damping out pressure oscillations.

Using the Rhie-Chow interpolation (6.30) for all face velocities in the pressure correction equation (6.27) with the general form (6.29) results in the final discretized pressure correction equation.

The algorithm for one time step of the SIMPLEC pressure-velocity coupling scheme is summarized in Algorithm 12.

Algorithm 12 Iteration procedure for the implemented SIMPLEC algorithm.

1. Guess an initial pressure field, and calculate pressure gradients

while not converged **do**

2. Store source terms for momentum equations
3. Assemble u, v and w momentum equations according to (6.17), store $a_{P,Sum}$ for each cell
4. Solve u, v and w momentum equations with an iterative solver (GMRES/BiCGStab)
5. Assemble the pressure correction equation according to (6.27)
6. Solve pressure correction equation with an iterative solver (CG)
7. Correct pressure and update pressure gradients according to (6.18) and (6.6)
8. Correct u, v and w velocity fields according to (6.18)
9. Calculate errors and check for convergence

end while

10. Update old velocities and pressures
11. Continue to the next time step with the current pressure field as initial pressure guess

6.1.9 Motivation and discussion of the used discretization schemes and methods

Mentioned earlier in this section was that a fully implicit time scheme was used when discretizing the temporal term. To motivate this choice let us briefly discuss other common choices. The explicit method also discussed puts a severe restriction on the maximum time step possible to ensure stability, being proportional to $\Delta t \propto \Delta x^2$ [44]. The limitation originates from that one want a positive coefficient for the temporal velocity from the previous time step. Even the Crank-Nicholson scheme suffers from such a drawback, but with a less restrictive proportionality constant. Note that both conditions are depending on Δx^2, implying that an increased accuracy in spatial discretization comes with a expense in a decrease of time step. These methods are consequently not well suited for general purpose codes, but might be beneficial in some applications, for example where objects are moving very fast requiring such a small time step for this reason. However, there also exist higher order explicit time schemes which are more accurate and stable, but with higher memory requirements and complexity introduced. The implicit scheme on the other hand is unconditionally stable, meaning that large time steps can be taken and still retaining stability. On the other hand it is only first order accurate, implying that a small time step is needed for good accuracy. The implicit scheme is nevertheless well suited for codes aimed at general unsteady problems.

As one might have noticed from the previous work chapter. Many existing reports are all using some kind of explicit scheme, leading to that the usage of an implicit time scheme is one factor that makes the work in this thesis significantly different. The explicit time scheme is less involved than the implicit, no linear system needs to be neither assembled nor solved. One can march forward in time by solving a single equation for

each momentum equation instead of having to solve a large system of equations as in the implicit case. The discretization of the momentum equations, in contrast to the pressure Poisson equation, generally does not produce a matrix either symmetric or positive definite, making the use of a relatively simple method such as CG unsuitable.

Implemented and tested convective schemes in this thesis are the first order upwind-, first order downwind-, and the Central scheme. These proved to be sufficient during the tests and simulations performed, but one can also think of higher order convective schemes, such as the second order upwind scheme. However, these require velocity values implemented from nodes further away than nearest neighbor, requiring larger discretization stencils, i.e. more non-zero elements in the resulting matrix. Which is also the main reason why such schemes were not implemented. The same broadening of stencil applies if higher order schemes in the discretization of the diffusive terms (and times) is used. Which is why such schemes were not implemented either. Implementing higher order schemes could be a topic for future research.

The pressure-velocity coupling scheme in this thesis was chosen to be SIMPLEC. There is no simple answer as to which of SIMPLE, SIMPLEC and SIMPLER pressure coupling schemes is the best [44]. SIMPLER and SIMPLEC introduce some extra complexity, but there is no guarantee that their convergence rates are improved compared to SIMPLE. Even though they often do. The SIMPLE, SIMPLER and SIMPLEC schemes are robust, and since they do not end until convergence, are always within some error bound. The drawback of the iterative procedure is increased simulation time. A limitation of the convergence rate of SIMPLE and SIMPLEC is that the velocity fields do not longer satisfy the momentum equation after correcting pressure. An algorithm which improves on this point is the PISO scheme [49], which performs a neighbor correction and skewness correction in each iteration in addition to those of SIMPLE and SIMPLEC. In the commercial general purpose flow solver Fluent's users manual [50], PISO is recommended for transient flow problems, especially when the time step used is large. However PISO introduce some increased complexity by taking into account neighbor interaction requiring the solution of additional systems of equations. In general textbooks such as [44], no absolute conclusion of which of the SIMPLE, SIMPLEC, or PISO algorithms that is best for general steady state problems is made. However for unsteady problems [49] showed that the temporal accuracy of the PISO algorithm's predictor corrector step is Δt^3 and Δt^4 for momentum and pressure respectively. This in turn makes a single iteration step enough for the majority of all applications, since often the time discretization schemes errors are of lower order. Thus, PISO becomes a non-iterative method requiring only one iteration. Interesting future research could be comparing the use of PISO and SIMPLEC in the code.

6.2 Boundary conditions and their treatment

Boundary conditions can be treated in different ways and affect the properties of the resulting linear systems and assembly routines. In CFD, one usually describes boundary conditions from their physical property, while in mathematics one usually speaks of

general forms of boundary conditions. In the collocated cell centered grid arrangement used in this thesis, the boundary lies on the faces of the control volumes. Ghost nodes can be introduced, lying at the imaginary cell on the other side of the boundary, see Figure 6.3. This leads to a neat way of folding the boundary nodes into the linear system by modifying the RHS and the coefficients of the coefficient matrix for neighboring nodes. For Dirichlet and Neumann boundaries, only the coefficients at the cell center and right hand side is modified, preserving symmetry if any.

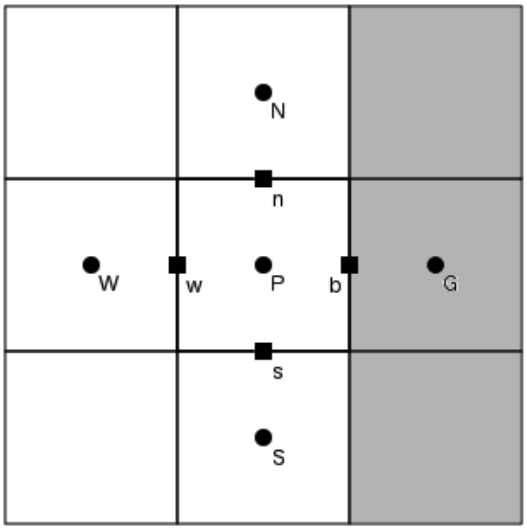


Figure 6.3: A representation of an east boundary in 2D. The white squares represent cells in the finite volume discretization, and the grey area is outside the computational domain. The ghost node on the other side of the boundary is denoted by G and the property on the boundary by b.

6.2.1 Robin boundary condition

Let us denote the property being evaluated across the boundary by ψ, then the general Robin boundary condition is given by

$$A\psi_b + B\frac{\partial \psi}{\partial n} = C. \tag{6.31}$$

Choosing $A = 1$, $B = 0$ results in a Dirichlet boundary and $A = 0$, $B = 1$ in a Neuman boundary. Using first order differencing for the derivative and linear interpolation of the value at the face, an expression for the value at the ghost node can be obtained as

$$\psi_G = \frac{C - \psi_P \left(\frac{A}{2} - \frac{B}{\Delta x}\right)}{\frac{A}{2} + \frac{B}{\Delta x}}, \tag{6.32}$$

which ensures the boundary condition is being satisfied. Now the coefficient for the imaginary node at the other side of the boundary can be assembled like any other node,

and then folded into the coefficient of the node at the cell center and the right hand side. The corrections to be made at the end of assembly can be summarized as

$$a_P = \overline{a_P} - \overline{a_G}\frac{\frac{A}{2} - \frac{B}{L}}{\frac{A}{2} + \frac{B}{L}}, \tag{6.33}$$

$$b = \overline{b} - \overline{a_G}\frac{C}{\frac{A}{2} + \frac{B}{L}}, \tag{6.34}$$

$$a_G = 0, \tag{6.35}$$

where overline variables denote the values before correction.

6.2.2 Extrapolation boundary condition

The extrapolation boundary condition describes an extrapolation of the property value across the boundary

$$\frac{\partial \psi_b}{\partial n} = \frac{\partial \psi_w}{\partial n}. \tag{6.36}$$

Applying a first order discretization on the derivatives, the value at the ghost node can be expressed as

$$\psi_G = 2\psi_P - \psi_W. \tag{6.37}$$

In the same way as for the Robin boundary condition, the boundary condition is folded into the coefficients of its neighbors and right hand side, resulting in a correction of the form

$$a_P = \overline{a_P} + 2\overline{a_G}, \tag{6.38}$$

$$a_W = \overline{a_W} - \overline{a_G}, \tag{6.39}$$

$$b = \overline{b}, \tag{6.40}$$

$$a_G = 0. \tag{6.41}$$

6.3 Properties and solution methods for the resulting linear systems

The discretized momentum equation (6.17) is non-linear. It was discretized such that the coefficients are dependent on the velocity from the previous time steps and previous SIMPLEC iterations, leading to a non symmetric system. When solving such a system we need to choose a solver algorithm suitable for such matrices. This makes CG unsuitable, leaving us with GMRES and BiCGStab. BiCGStab proved to be most efficient for matrix types relevant to this thesis, see Figure 5.2, both regarding memory requirements and solution time. However, in a general and robust code it could be wise to fall back to GMRES when, if ever, BiCGStab fails.

The pressure equation is symmetric and positive definite, at least for Neumann and Dirichlet boundaries. CG was discovered to be the fastest and most memory efficient choice, see Figure 5.2.

6.4 CUDA considerations for the flow solver implementation

Major CUDA optimization techniques have already been discussed in Chapter 2. As well as difficulties and differences to serial CPU implementations. Recall that the main difference between CPU and GPU lies in the SPMD structure of the GPU programs, together with higher requirements of manually obtaining good data locality. Threads within blocks can communicate via shared memory and barrier synchronization but threads from different blocks cannot. This leads to that all variables read by all threads need to be known before a call. Subsequently the GPU code must be split into several smaller programs with synchronization on the CPU side between the calls. Note though, that the memory resides on the GPU during execution of the program, and that the CPU mostly acts as flow control, possibly doing computations parallel to the GPU kernels. The flow of the Navier-Stokes solver specified by GPU kernels is detailed in Algorithm 13.

Algorithm 13 Process flow of GPU Navier-Stokes solver specified by GPU kernels

1. Allocate GPU memory and initialize variables.
2. GPU_kernel: Connect grid to its neighbors using a cell structure.

while not converged **do**

3. GPU_kernel: Assemble u, v and w momentum matrices and corresponding RHS's, fold boundary conditions into system, store $a_{P,Sum}$ for each cell.
4. Call iterative solver: solve u, v and w momentum equations (Calculated on GPU through an iterative method).
5. GPU_kernel: Assemble the pressure correction matrix and RHS, fold boundary conditions into system.
6. Call iterative solver: solve the pressure equation (Calculated on GPU through an iterative method).
7. GPU_kernel: Correct pressure using an axpy operation.
8. GPU_kernel: Calculate pressure gradient.
9. GPU_kernels: Correct u, v and w velocity fields according to (6.18), using modified axpy operations.
10. GPU_kernel: Calculate errors and check for convergence using several kernels involving reduction operations: finding maximum of array, summing elements of an array.

end while

11. GPU_kernel: Update old velocities and pressures
12. Continue to the next time step with the current pressure field as initial pressure guess

Note that all calculations of the implementation are implemented on the GPU. No costly memory transfers between the CPU and the GPU are required. All parts can effectively be parallelized on the GPU.

The implemented finite volume method discretization adopted in this thesis is memory

intensive. For most calculations, a property value of a neighboring node is read with only a single arithmetic operation performed on it. For this reason, the main optimization effort was put on optimizing memory throughput. Recall that to achieve full global memory bandwidth coalesced memory fetching need to be used. For this reason, each thread was assigned to handle/assemble/calculate one row of the matrix or element of a vector corresponding to one cell. The threads were assigned to consecutive cells. Cell values for different properties were stored in the same order for all properties: consecutive cell properties in consecutive positions in memory, without other cell properties in between. In this way complying to the useful advice: use struct of arrays, not arrays of structs.

Further improvements on memory limitations can be done by reducing the number of accesses to global memory. This was done by using the lower level memory spaces and reusing data. Registers were used for properties being accessed or read multiple times per thread to as large extent as possible. But as has been mentioned, using more registers than available on an SM reduces the number of threads launched by it. To limit the number of registers used, the registers were reused within a thread where possible. The constant memory was used to store constants, such as density and time step etc., offloading registers and speeding up accesses. The constant memory was not used fully, some potential for improvement lies in making more use of it.

Shared memory and texture memory were not used extensively in the assembly routines. It is in utilizing these memory spaces better the highest speedup potential for future work lies. The potential for shared memory lies in using the technique called tiling. This technique was used by [26], but without comparison to a non-tiled algorithm. For implementation of tiling for the assembly routines, the idea is to let each block of threads be responsible for one spatial cube of cells in the spatial domain. The great benefit is that properties used by neighbors only need to be fetched from global memory once for the whole thread block and can then be fetched from the shared memory. An inner node property only need to be fetched once instead of up to seven times, one for each neighboring cell and the middle cell itself. Due to shared memory size limitations per SM, the number of variables beneficial in storing in shared memory is limited. The other caution to consider with shared memory is bank conflicts, but this is not expected to be an issue for the assembly routines. Texture memory or in modern GPUs, read only memory, can speedup in an analogous way since textures are cached effectively for spatial locality. In [28], an average speedup of 1.5 times when using texture memory for finite difference stencils is mentioned.

Reusing data is usually implemented by letting the same thread calculate properties for multiple elements or cells. In this thesis, it was used by correcting for boundary conditions in the same kernel as the general assembly was being performed. In this way being able to reuse register variables and reducing memory fetches from global memory.

6.5 Results and validation

The implemented Navier-Stokes solver has been validated and performance tested by simulating a lid-driven cavity problem. The CPU used is an Intel Xeon E5-2650 and the

GPU used is a Nvidia Geforce GTX 660 Ti, see Table 2.1 for hardware details. Double precision floating point numbers have been used for both GPU and CPU computational results presented in this chapter.

6.5.1 Lid-driven cavity validation

A common benchmark and validation case for CFD code is the lid-driven cavity case. It models an idealized recirculation flow with applications to environmental modeling, geophysics and industry. The reason for its popularity as validation case is due to its simple domain description and that it nevertheless exhibits fluid flow behavior of complex nature, such as counter rotating vortices.

The flow is characterized by its Reynolds number, defined by

$$RE = \frac{vL\rho}{\mu}, \tag{6.42}$$

where L is the length of the domain, v velocity, ρ density, and μ dynamic viscosity. In 3D, the flow is modeled by a cubic regime surrounded by 6 walls. One of the walls is set to a constant velocity while the others are set to be stationary. Mathematically, Dirichlet boundary conditions are used for the velocity at all walls, while Neumann is used for the pressure. The interested reader is referred to [51], where the lid-driven cavity flow is extensively discussed. A good source for tabular data for flows with various Reynolds numbers of the lid-driven cavity case is [52]. This data set is commonly used for validating and benchmarking CFD code. The data was used also in this thesis work to validate the flow solver.

The 2D lid-driven cavity case was simulated and validated to the data provided by [52] for laminar flow with Reynolds numbers of $RE = 100$, $RE = 400$ and $RE = 1000$. A 2D grid of 129x129 grid points was used. The velocity stream lines for $RE = 1000$ can be seen in Figure 6.4, color coded by velocity magnitude. Observe the main vortex in the middle and the small ones in the lower corners. The validation plots can be seen in Figure 6.5. The validation is seen to have satisfactory precision for all different validated Reynolds numbers.

6.5.2 Performance results

The performance of the implemented GPU Navier-Stokes solver was evaluated against CPU implementations for the lid-driven cavity case. The GPU version was compared to a CPU ported version of it in 2D, and further to a general purpose flow solver in 3D. The full simulation results presented correspond to the simulation of 100 time steps of equal step size.

To have a truly fair comparison of the GPU implementation to an equivalent optimized CPU version for all aspects of the program, the GPU flow solver was ported to an almost identical OpenMP threaded CPU implementation in 2D.

Performance comparison between the CPU and GPU implementations for different sizes of a 2D square lid-driven cavity domain can be seen in Figure 6.6. In the figure, the

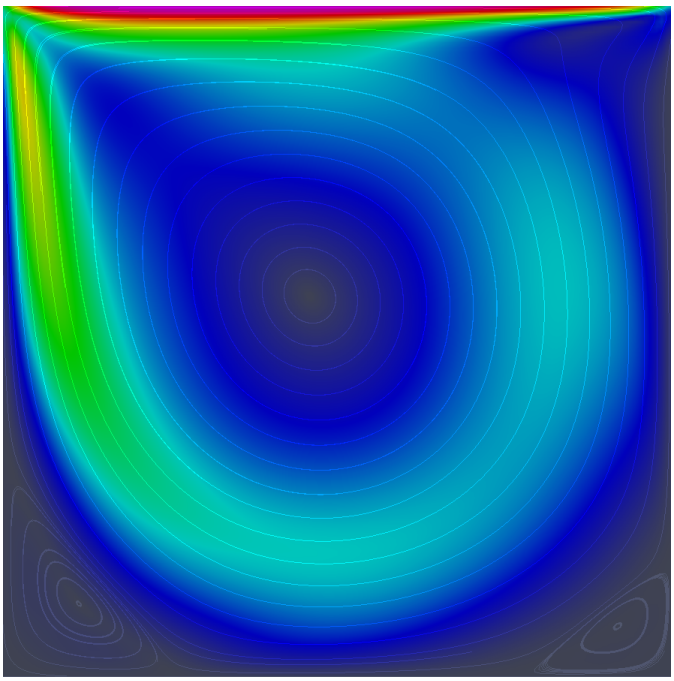
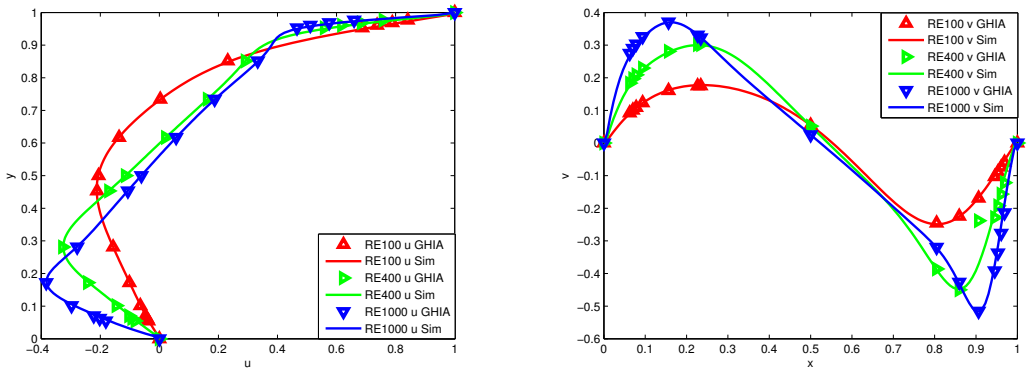


Figure 6.4: Stream lines color coded by velocity magnitude for the 2D lid-driven cavity case with $RE = 1000$.



(a) u velocity along a vertical line through the geometrical center of cavity.

(b) v velocity along a horizontal line through the geometrical center of cavity.

Figure 6.5: Validation of the 2D lid-driven cavity case with Reynolds numbers of 100, 400 and 1000, respectively, to the data of [52].

speedup of the total solution time and assembly time is shown. The time distribution of the total solution time divided between the time required for iterative solvers and assembly routines respectively can be seen in Figure 6.7, while the time distribution between the different assembly routines and time distribution between different linear algebra solvers respectively is shown in Figure 6.8. It is seen that the speedup increases as the problem size grows larger. At small problem sizes, the GPU is not fully utilized and the CPU's larger caches effectively stores most parts of the data on high speed low latency memories. As the problems grows larger the speedup increases. It is unknown why the speedup grows more for full simulation, i.e. the iterative methods, than for the assembly routines, but the details of implementation of the MKL library remains unknown and the assembly routines are not fully optimized. As is seen in Figure 6.7, the time required for assembly routines is negligible compared to solving the resulting linear systems. The speedup for the assembly routines is up to a factor of slightly less than six.

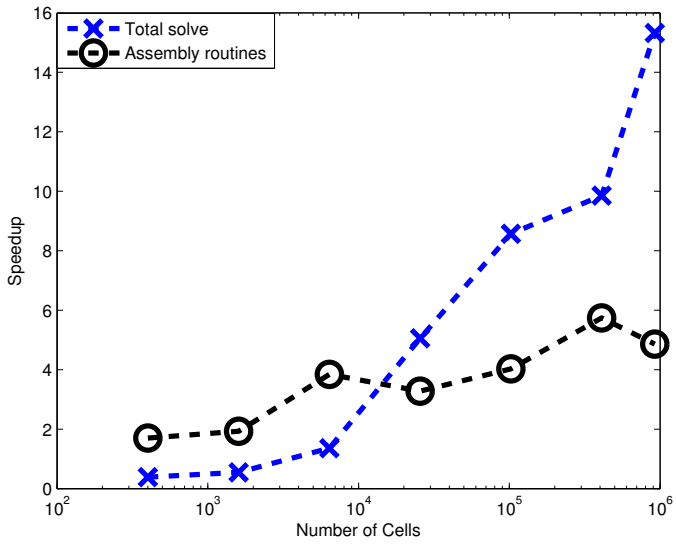


Figure 6.6: Speedup of the GPU implementation compared to the OpenMP threaded CPU implementation. Speedup for solving the whole problem is plotted in blue crosses, while the speedup for the time required for assembly routines is plotted with black circles.

The GPU implementation was further evaluated against a general purpose flow solver developed at FCC, IBOFlow, with the purpose of giving an insight into the expected speedup if the whole Navier-Stokes solver was to be implemented on the GPU. Two choices of sparse linear systems solvers in IBOFlow were tested, an algebraic multigrid method (AMG) and the MKL library solvers. The motivation for choosing AMG was to see the speedup compared to one of the fastest methods available for CPUs. MKL was tested to see the speedup of equivalent GPU-CPU implementations. The evaluated test case was the lid-driven cavity on a cubic 3D domain of different sizes.

In Figure 6.9 the speedup of the implemented GPU solver over a CPU solver with

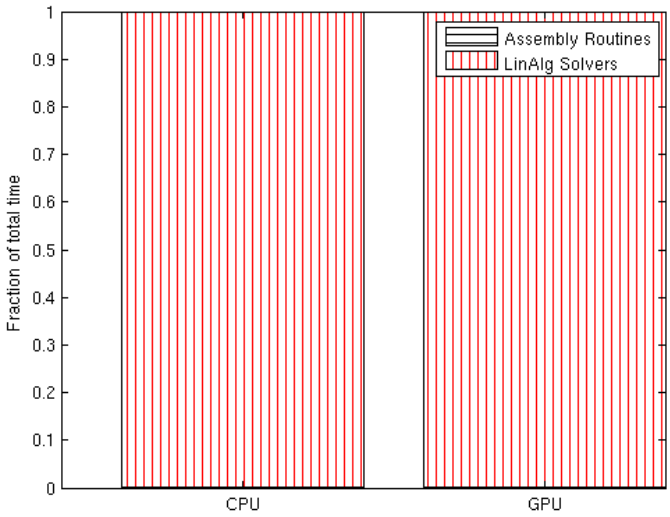
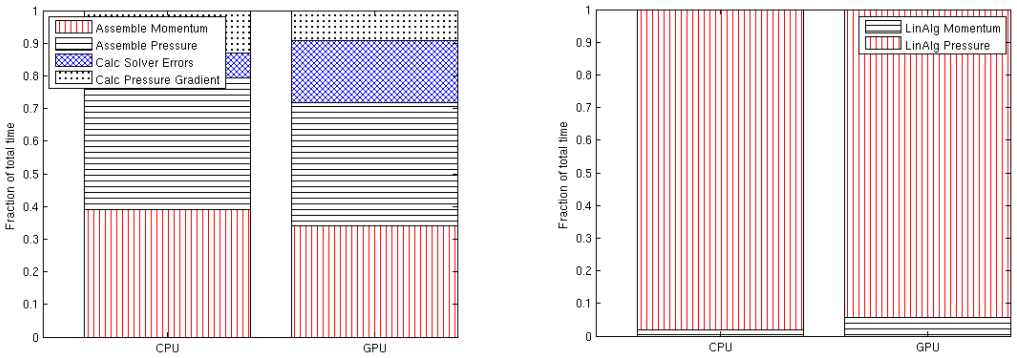


Figure 6.7: Fraction of total time required for the linear algebra solvers and assembly routines, respectively, for the lid driven cavity case in 2D. The grid size tested is of $640^2 = 409600$ cells.



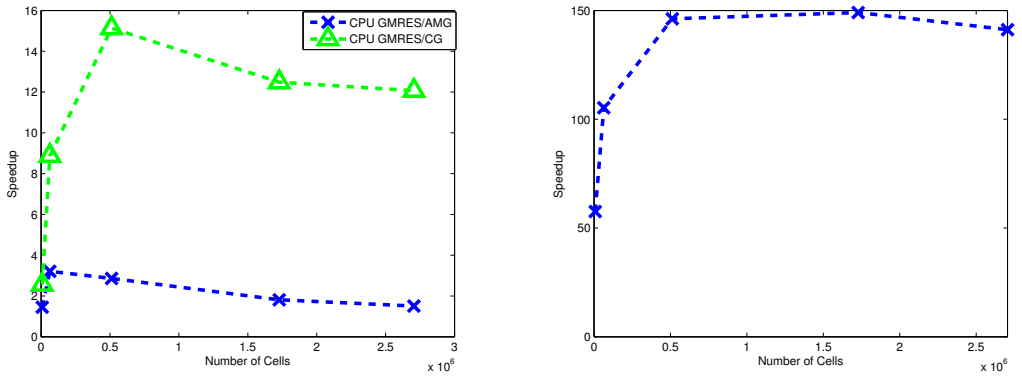
(a) Fraction of total time required for assembly routines divided between the different assembly routines.

(b) Fraction of total time required for the Linear Algebra solvers divided between the different routines.

Figure 6.8: Time distribution of assembly routines and linear algebra routines for the lid driven cavity problem of grid size $640^2 = 409600$ cells.

GMRES/AMG sparse system solver and a CPU solver with GMRES/CG sparse system solver can be seen, respectively. To clarify, GMRES/AMG refers to that MKL's GMRES method was used to solve the momentum matrices while an AMG method was used to solve the pressure matrix system. The speedup for the assembly routines are up to an astonishing factor of 150 times, however, the CPU implementation for those routines was

not parallel. The overall speedup is more moderate, especially when the asymptotically much more effective AMG method for solving the Poisson pressure equation in IBOFlow.



(a) Speedup of the total solution time compared to a CPU implementation with MKL CG and an AMG method respectively.

(b) Speedup of the time required for assembly routines of the GPU implementation compared to the CPU solver.

Figure 6.9: Speedups of the implemented GPU flow solver compared to a general purpose flow solver for the 3D lid-driven cavity problem.

A natural question, looking at Figure 6.9, is how the total solution time is distributed between different operations. To identify bottlenecks in the implemented GPU solver, the distribution of execution time between different routines for the case of a domain with 120^3 cells was tested. In Figure 6.10, the time distribution between linear algebra solve time and assembly time can be seen. In Figure 6.11, the time distribution between the assembly routines and linear algebra solvers can be seen.

For the GPU solver, the time required for assembly routines is in practice negligible. Further, the time required for solving the momentum equation is small compared to that of solving the pressure Poisson equation. Speeding up the solution time for the pressure Poisson equation is the key to speeding up the GPU Navier-Stokes solver.

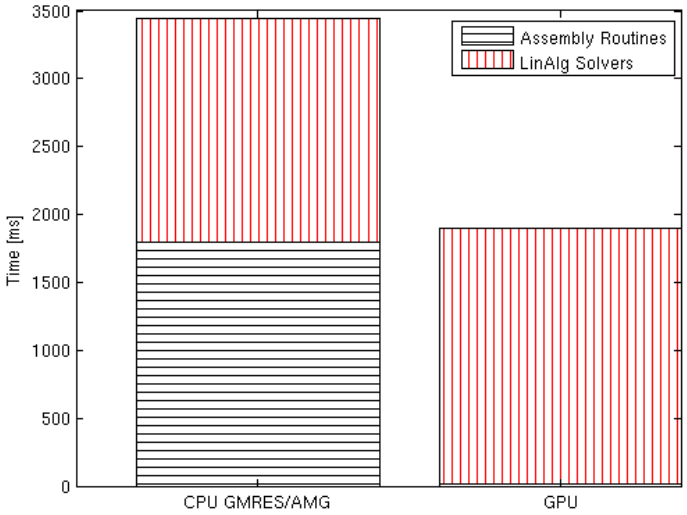
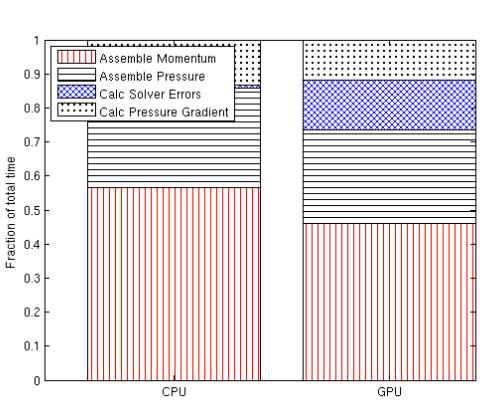
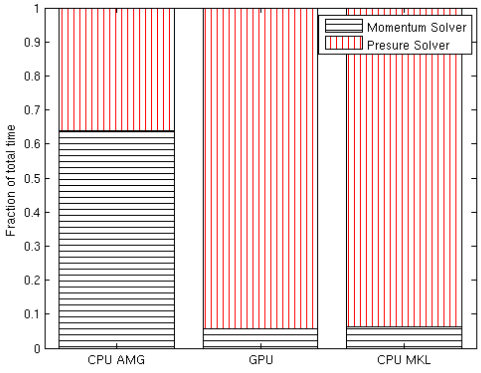


Figure 6.10: Time distribution between linear algebra routines and assembly routines for the general purpose CPU solver with AMG solver for pressure and the GPU solver respectively. The tested case is the 3D lid-driven cavity problem on a square domain of size $120^3 = 1728000$.



(a) Time distribution of assembly routines for the GPU and CPU.



(b) The distribution of linear algebra solvers for GPU, CPU with CG for pressure and CPU with AMG for pressure.

Figure 6.11: Time distribution of the total time required for different assembly routines and linear algebra solvers respectively. The tested case is the 3D lid-driven cavity problem on a square domain of size $120^3 = 1728000$ cells.

Chapter 7

Discussion, conclusion and outlook

7.1 Discussion

7.1.1 Complementing discussion of results

Let us address the reason for the huge speedup of 150 for the assembly routines of the GPU solver and the general purpose flow solver tested, IBOFlow, seen in Figure 6.9. These numbers need to be considered with a few things in mind. The GPU implementation is more optimized, whereas the CPU implementation has been developed with higher focus on clear and reusable code. The CPU implementation is serial, parallelizing it to 8 cores would give an estimated speedup of up to 8 times for large enough problems. The CPU assembly routines are more general, including the possibility of handling locally refined grids and more complex boundaries. Implementing these into the GPU solver would likely increase the time slightly, however, the expected increase in time is relatively small. Another thing to keep in mind, being an advantage for the GPU implementation, is that the GPU tested on is not particularly expensive compared to the CPU tested to. If memory is the only limiting factor, as is likely the case, then a GPU upgrade to an Nvidia Geforce Titan, which has twice the memory bandwidth over the GTX 660 Ti, would give an additional estimated speedup with a factor of two.

The curious reader might have asked himself/herself, why bother to implement a full Navier-Stokes solver on the GPU when most potential of achieving speedups comes from speeding up the iterative solvers? This is a legitimate question, and preliminary tests of this kind were done initially in the project. Recall the issue with slow CPU-GPU transfers mentioned in Chapter 2, this will likely slow down a heterogenous program. Therefore to evaluate the full potential, the whole Navier-Stokes solver was implemented on the GPU.

7.1.2 Performance bottlenecks and possible solutions

From Figure 6.10 and Figure 6.11 it is obvious that to further improve the performance of the GPU solver significantly, the solution of the pressure Poisson equation must be speeded up. Several possible ways of achieving this exist. The initial pressure guess for every SIMPLEC iteration could be better. Another is assembling a matrix with better convergence properties, which might be achieved by using another pressure-velocity coupling scheme. Other methods such as PISO, SIMPLE and SIMPLER have been discussed and also the projection method Fractional Step Method, FSM. All of these could be evaluated for this reason, even though the accuracy of solution for the FSM should be worse than for the others. Further, since SIMPLEC is iterative and PISO, for example, is not, the number of pressure and momentum equations needed to be solved per time step might be reduced by using such a method.

The most obvious way to speed up the solution of the pressure Poisson equation is by looking into the solution method used for the linear system. One way could be to look into alternative preconditioners. A popular preconditioner, the ILU0, was tested as a part of this thesis along with the least square polynomial preconditioner, the latter which have reported good performance in the literature [18, 42]. Neither of them was found to be particularly efficient for the tested matrices and solvers. Thus the potential to find good speedups using traditional preconditioners are likely limited. One thing to be tested, though, could be using variable preconditioning, where the preconditioner is allowed to change between iterations.

Best potential for improvement of the solution is by using some kind of multigrid method, either as a preconditioner or as a solver. Multigrid methods were described in Chapter 5 and can be optimal, of order $O(n)$, for solving sparse linear systems. It was seen that for the CPU, the AMG multigrid method was superior to the iterative methods, Figure 6.9. Further efficient examples of multigrid implementations on GPUs exist in the literature, [22] and [23]. The first reports of a simple implementation, mainly demonstrating multigrids potential on the GPU, while the second could serve as inspiration for future developments. Multigrid methods are involved and hard to develop for complex domains, and requires parameter fine tuning to work efficiently. Furthermore the author does not know of any efficient commercial implementation, hinting that writing such a method is probably a challenge.

To further speed up the iterative methods with a small amount, kernel fusion could achieve speedups if implemented successfully [21]. A technique which could be considered for fine tuned optimization in the future. A related method is to enable the possibility of solving several right hand sides simultaneously. For solving the momentum equations in the discretized 3D Navier-Stokes equations, it is required to solve the system of equations for 3 right hand sides using the same coefficient matrix. Specifically, the SpMV can allow for multiplying many vectors at the same time.

7.1.3 Further GPU techniques and considerations

The techniques used to optimize the assembly routines of the Navier-Stokes solver were mentioned along with possible future optimizations. Due to the spatial locality of access patterns, each cell requires properties from neighboring cells, there is a potential speedup in using either or both of tiling and texture/read only memory. These optimizations are interesting from a GPU perspective, but much less interesting from the perspective of speeding up the Navier-Stokes solver. It was seen in Figure 6.10 that the time required for assembly routines is negligible compared to the time required for solving the pressure Poisson equation.

An issue not explicitly addressed earlier is how large problems that can be solved on the implemented GPU Navier-Stokes solver using only a single GPU. The memory limit was hit when using 139 x 139 x 140 = 2704940 cells, with a GPU memory of 3GB. There exist GPUs on the market with 6GB memory. If even larger simulations are desirable, a multi-GPU system need to be used. The newer CUDA GPUs have multi-GPU support and the GPU-GPU communication can be done with an MPI interface. Also, GPUs in the same system can directly access each others memory through high-bandwidth connections. A flow solver on a multi-GPU system has been implemented in [27], reporting reasonable scaling for several GPUs. Since GPU kernels are naturally massively parallel and the model used is scalable, porting a single GPU application to multi-GPU should be relatively straight forward. Especially, the algorithms often do not need to be fundamentally changed. Splitting of vectors and matrices for the iterative solvers likely requires some coordination though.

7.1.4 Single vs. double precision floating point numbers and bandwidth limitations

Another question relevant to GPGPU applications is the use of single and double precision. On graphics oriented GPUs, such as the GTX 660 Ti used for the tests in this thesis, the single precision processors, and theoretical maximum GFLOPS, greatly outnumber the number of double precision processors and theoretical maximum GFLOPS. Whereas on computationally oriented GPUs the single and double precision processors are of the same number (these GPUs are more high end and more expensive though). The question arises whether the number of double precision processors is a limiting factor on graphics oriented processors such as the Nvidia Geforce GTX 660 Ti, or if it still is memory bandwidth limiting the performance. If the the number of double precision processors is the limiting factor, then using a computationally oriented GPU would increase the performance of the solver. Preliminary tests for using single precision with the solvers have been performed, but the loss in accuracy is sometimes too significant. Often the overall time when using single and double precision is about the same. This is due to that the reduced accuracy in single precision makes the single precision solvers require more iterations to converge than the double precision ones. Concluding that using single precision is not an alternative for the target application. Observe though, that if the allowed error criteria is lowered, using single precision might be beneficial.

The limiting factor in the iterative solvers was studied by two tests, timing the matrix-vector kernel for double and single precision and solving a complete linear system, on two different test matrices. Timings for single and double precision can be seen in Table 7.1. The timing for double precision is slightly less than double that of the single precision in all cases, except for the HYB format matrix-vector operation in the case of 63604 elements. Why this differs might be due to efficient caching of small matrices, but the larger cases are more interesting. An almost doubling of time for double precision is expected since double precision requires 8 bytes of storage while single precision requires 4 bytes. This means that the required memory bandwidth is twice for double precision as compared to single precision. The iterative solvers seem to be bandwidth bound, meaning that using a graphics card with increased memory bandwidth would likely speed up the iterative solvers by the same amount.

Table 7.1: Timings for single and double precision cuSPARSE hybmv operation (SpMV operation in the HYB sparse format) and a solve phase for the CG solver on a CFD pressure matrix. Notice that the timings for double precision is almost double that of the timings for single precision.

Operation and # cells	**Precision**	**Time [ms]**	**Iterations**	**time/iteration**
hybmv 63604	single	1.3e-4	N/A	N/A
hybmv 63604	double	3.7e-4	N/A	N/A
hybmv 1289454	single	1.69e-3	N/A	N/A
hybmv 1289454	double	3.1e-3	N/A	N/A
CG Solve 63604	single	0.067	359	1.87e-4
CG Solve 63604	double	0.0103	424	2.43e-4
CG Solve 1289454	single	1.204	575	2.10e-3
CG Solve 1289454	double	2.31	589	3.9e-3

7.2 Conclusion and outlook

In this thesis work, the applicability of GPUs for solving the incompressible Navier-Stokes equations has been assessed. The sparse linear algebra iterative methods CG, GMRES and BiCGSTAB have successfully been implemented on the GPU. The sparse linear algebra preconditioners least square polynomials and diagonal scaling have been implemented on the GPU. The 3D incompressible Navier-Stokes equations have been discretized, implemented and solved on the GPU using CUDA.

The suitability of different sparse formats for the iterative solvers has been evaluated. For the kind of matrices produced in this thesis, the HYBrid format proved best suited, see Table 4.1 and 4.2. The iterative solvers implemented performed better than other GPU accelerated open source libraries, see Figure 5.5. The solvers also achieved an average speedup factor larger than six compared to the parallel CPU solvers from the

MKL library on an Intel Xeon-E5-2650, see 5.4.

The implemented Navier-Stokes solver was successfully validated on the lid-driven cavity problem to the tabular data of Ghia et al. [52]. The GPU implementation achieved an average speedup of a factor six for the assembly routines and up to fifteen for the full simulation, compared to an identical OpenMP threaded CPU implementation written in this thesis work. Compared to a commercial general purpose CPU CFD code, the implemented GPU solver achieved an average total speedup of over a factor twelve when equivalent linear algebra solvers were used by the two programs. When an algebraic multigrid method was used for solving the pressure equation on the CPU, an average total speedup of 2.5 was achieved. In the comparisons, an Nvidia Geforece GTX 660 Ti GPU, 300$, and an Intel Xeon E5-2650 8-core CPU, 1000$, was used.

The time consuming part of the solution phase for the implemented solver was identified to be solving the pressure Poisson equation using the CG algorithm. Time required for assembly routines, and for solving the momentum equations are negligible in comparison. It is in trying to improve the solve time for the pressure Poisson equation the potential for speedup for future work lies. Experimenting more with preconditioners, implementing a multi-grid method on the GPU, and using the PISO coupling scheme instead of SIMPLEC were techniques discussed for achieving this.

Furthermore the Navier-Stokes solver could be extended to be more general in the future. Attractive features would be the possibility of a locally refined grid and to include immersed boundaries for handling complex geometries and objects in the solution domain.

The GPU has been evaluated and proven to be able to achieve significant speedups of almost an order of magnitude over CPUs for the iterative solvers as well as for the assembly routines of the Navier-Stokes equations. The scalable architecture of CUDA GPUs will likely make the implemented GPU solver amenable to larger speedups when new generations of GPUs hit the market.

Bibliography

[1] Titan supercomputer. http://www.olcf.ornl.gov/titan.

[2] IBOFlow - Immersed Boundary Octree Flow Solver. http://www.iboflow.com.

[3] Andreas Mark, Erik Svenning, and Fredrik Edelvik. An Immersed Boundary Method for Simulation of Flow with Heat Transfer. *International Journal of Heat and Mass Transfer*, Accepted for publication 2012.

[4] Andreas Mark, Rundqvist Robert, and Fredrik Edelvik. Comparison Between Different Immersed Boundary Conditions for Simulation of Complex Fluid Flows. *Fluid Dynamics and Material Processing*, 7(3):241–258, 2011.

[5] Andreas Mark and Berend van Wachem. Derivation and Validation of a Novel Implicit Second-Order Accurate Immersed Boundary Method. *Journal of Computational Physics*, 227(13):6660–6680, 2008.

[6] FCC - the Fraunhofer Chalmers Research Centre for Industrial Mathematics. http://www.fcc.chalmers.se.

[7] NVIDIA cuSPARSE. cublas libraries.

[8] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[9] Kandrot Edward Sanders Jason. *CUDA By Example: An Introduction to General-Purpose GPU programming*. Addison-Wesley, November 2010.

[10] Mark Harris and Godekke Dominik. Gpgpu.org. http://gpgpu.org.

[11] BrookGPU. http://graphics.stanford.edu/projects/brookgpu/index.html.

[12] Sh metaprogramming language. http://www.libsh.org.

[13] Jack Foertter, Fernanda & Wells. Accelerating science and engineering with titan, the world's fastest supercomputer, 2013. Speech at GPC 2013.

[14] Hyesoon Kim, Vuduc Richard, Baghsorkhi Sara, Choi Jee, and Hwu Wen-mei. *Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)*. Synthesis lecture on computer architecture. Morgan & Claypool publishers, November 2012.

[15] NVIDIA. *CUDA C PROGRAMMING GUIDE v5.0*. NVIDIA, October 2012.

[16] Mark Harris et al. Optmizing parallel reduction in cuda. *NVIDIA developer technology*, 2, 2007.

[17] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[18] Ruipeng Li and Yousef Saad. Gpu-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

[19] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies. *IBM Reserach Report, RC24704 (W0812-047)*, 2008.

[20] Maxim Naumov. Parallel incomplete-lu and cholesky factorization in the preconditioned iterative methods on the gpu.

[21] J Filipovič, M Madzin, J Fousek, and L Matyska. Optimizing cuda code by kernel fusion—application on blas. *arXiv preprint arXiv:1305.1183*, 2013.

[22] Jeroen Molemaker, Jonathan M Cohen, Sanjit Patel, and Jonyong Noh. Low viscosity flow simulations for animation. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 9–18. Eurographics Association, 2008.

[23] Markus Geveler, Dirk Ribbrock, Dominik Göddeke, Peter Zajac, and Stefan Turek. *Efficient finite element geometric multigrid solvers for unstructured grids on GPUs*. Techn. Univ., Fak. für Mathematik, 2011.

[24] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.

[25] Andrew Corrigan, Fernando F Camelli, Rainald Löhner, and John Wallin. Running unstructured grid-based cfd solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids*, 66(2):221–229, 2011.

[26] T. Brandvik and Pullann G. Acceleration of a 3d euler solver using commodity graphics hardware. In *46th AIAA Aerospace Sciences Meeting and Exhibit*, January 2008.

[27] J. C. Thibault and I. Senocak. Cuda implementation of a navier-stokes solver on multi-gpu desktop platsforms for incompressible flows. In *47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aeruspace Exhibit*, January 2009.

[28] J Cohen and M Jeroen Molemaker. A fast double precision cfd code using cuda. *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, pages 414–429, 2009.

[29] Christopher P Stone, Earl PN Duque, Yao Zhang, David Car, John D Owens, and Roger L Davis. Gpgpu parallel algorithms for structured-grid cfd codes. In *Proceedings of the 20th AIAA Computational Fluid Dynamics Conference*, volume 3221, 2011.

[30] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2004.

[31] PR Rinaldi, EA Dari, MJ Vénere, and A Clausse. A lattice-boltzmann solver for 3d fluid simulation on gpu. *Simulation Modelling Practice and Theory*, 25:163–171, 2012.

[32] Yousef Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003.

[33] Li Ruipeng and Yousef Saad. CUDA ITSOL. http://www-users.cs.umn.edu/ saad/-software/.

[34] Dimitar Lukarski. PARALUTION project. http://www.paralution.com/.

[35] James W Demmel. *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics, 1997.

[36] Richard Barrett, Michael Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*. Number 43. Society for Industrial and Applied Mathematics, 1987.

[37] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 1. JHUP, 1993.

[38] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[39] Maxim Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu. Technical report, NVIDIA Technical Report, NVR-2011-001, 2011.

[40] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[41] Jocelyne Erhel, Guyomarc'H Frédéric, Yousef Saad, et al. Least-squares polynomial filters for ill-conditioned linear systems. 2001.

[42] Y Liang, Jim Weston, and Marek Szularz. Generalized least-squares polynomial preconditioners for symmetric indefinite linear equations. *Parallel computing*, 28(2):323–341, 2002.

[43] Intel math kernel (MK) library. http://software.intel.com/en-us/articles/intel-mkl.

[44] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An introduction to computational fluid dynamics: the finite volume method*. Prentice Hall, 2007.

[45] Andreas Mark. *The Mirroring Immersed Boundary Method - Modeling Fluids with Moving and Interacting Bodies*. PhD thesis, Chalmers University of Technology, 2008.

[46] Michael T. Heath. *Scientific Computing: An Introductory Survey 2nd ed.* McGraw-Hill, 2002.

[47] Suhas V Patankar and D Brian Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal of Heat and Mass Transfer*, 15(10):1787–1806, 1972.

[48] C. M. Rhie and W. L. Chow. Numerical study of the turubulent flow past an airfoil with trailing edge separation. *AIAA Jorunal*, 21(11):1525–1532, November 1983.

[49] Raad I Issa. Solution of the implicitly discretised fluid flow equations by operator-splitting. *Journal of Computational physics*, 62(1):40–65, 1986.

[50] Fluent Fluent. 6.3 user's guide. *Fluent Inc*, 2006.

[51] Ercan Erturk. Discussions on driven cavity flow. *International journal for numerical methods in fluids*, 60(3):275–294, 2009.

[52] U Ghia, KN Ghia, and CT Shin. High-re solutions for incompressible flow using the navier-stokes equations and a multigrid method. *Journal of computational physics*, 48(3):387–411, 1982.