UNIVERSITY OF THESSALY

DEPARTMENT OF MECHANICAL ENGINEERING

Diploma Thesis

Benchmark Heat Transfer and Rarefied Gas Flow Problems on Graphics Processing Units

by

Efstratios Kritikos

A Dissertation Submitted for the Partial Fulfillment of the
Requirements for the Degree of
Mechanical Engineering Diploma

February, 2019

Certified by the members of the Tripartite Dissertation Committee:

1st member	Professor Dimitris Valougeorgis
(Supervisor)	Professor in the Department of Mechanical Engineering
	University of Thessaly
2nd member	Dr. John Lihnaropoulos
	Instructor in the Department of Mechanical Engineering
	University of Thessaly
3rd member	Dr. Christos Tantos
	Postdoctoral Researcher in the Department of Mechanical Engineering
	University of Thessaly

To my parents;

my past, my present and

the dim light that I follow

Acknowledgments

First and foremost, I owe an everlasting thank you to Dr. Christos Tantos, for his admirable support, passion, dedication and profound time devotion on this diploma thesis to help me first to learn, and then put into practice, as well as cherish this long lasting journey. Furthermore, I would like to express my sincerest gratitude towards my supervisor, Professor Dimitris Valougeorgis, for his excellent tutelage and guidance through my research steps; his contributions to yield a satisfactory and valuable result. I am also indebted to Instructor John Lihnaropoulos for dedicating their time to read and correct this diploma thesis, as well as his assistance for any occurring technical computer problems.

Looking back I am certain that I could not have accomplished anything without the endless support from my family, Antigoni Tzonou, Michalis Kritikos and Sapfo Kritikou, through many frustrations and difficulties.

Finally, we gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.

Abstract

Many simulation and numerical analysis problems in the field of mechanical engineering require considerable computational time, in order to convergence to a solution. Currently, the continuous improvement of processor's performance has become moderately stagnant, due to physical constraints, such as the microscopic size of the transistors and the utilized materials. Consequently, the acquisition of solutions in reasonable timeframes is achieved by using parallel programming, according to which more than one processors are used by one or more computers connected over a network with each other. Many parallel programming models have been developed, of which perhaps the most promising one is CUDA, that allows a program to run on graphical processing units. In this diploma thesis CUDA model has been applied to benchmark problems, that greatly suffer from tremendous computational time.

Initially, a heat transfer problem through an L-shaped fin is examined, the geometry and magnitude of which is applied to cooling fins, especially of computer parts. Through the developed code, it is possible to investigate more complex shapes and various materials that are encountered in practice. Afterwards, the flow of a rarefied gas is studied between two parallel plated of infinite length and width (one-dimensional flow) and inside a duct of rectangular cross section (two-dimensional flow). This particular flow is known in the literature as Poiseuille flow.

The obtained results indicated high performance and acceleration of all parallel code executions, in comparison to the respective serial ones. Specifically, for the heat transfer problem the parallel code was up to 410 times faster than the serial code. Likewise, in the one-dimensional flow of a rarefied gas an acceleration of 46 times was succeeded, while

in the two-dimension flow up to 298 for double and 568 for single precision numbers. The extremely fast solving with CUDA has confirmed its impressive and immense usage on many important programming applications, as well as its continuous consolidation on many scientific and research fields.

Περίληψη

Πολλά προβλήματα προσομοίωσης και αριθμητικής ανάλυσης του πεδίου της μηχανολογίας απαιτούν σημαντικό υπολογιστικό χρόνο για την σύγκλιση σε λύση. Πλέον, η συνεχής βελτίωση της απόδοσης των επεξεργαστών έχει φτάσει σε τέλμα, λόγω φυσικών περιορισμών, όπως το μικροσκοπικό μέγεθος των τρανζίστορς και τα χρησιμοποιούμενα υλικά. Επομένως, η απόκτηση λύσεων σε λογικά χρονικά πλαίσια επιτυγχάνεται με τη χρήση του παράλληλου προγραμματισμού, σύμφωνα με τον οποίο περισσότεροι του ενός επεξεργαστές χρησιμοποιούνται από έναν ή περισσότερους υπολογιστές, οι οποίοι συνδέονται μέσω δικτύου μεταξύ τους. Πολλά μοντέλα παράλληλου προγραμματισμού έχουν αναπτυχθεί, εκ τον οποίων ίσως το πιο πολλά υποσχόμενο είναι η CUDA, που επιτρέπει την εκτέλεση ενός προγράμματος σε κάρτες γραφικών. Στη παρούσα διπλωματική το μοντέλο της CUDA εφαρμόστηκε σε προβλήματα, που υποφέρουν σημαντικά από τεράστιο υπολογιστικό χρόνο.

Αρχικά, το πρόβλημα της μεταφοράς θερμότητας εξετάζεται σε ένα πτερύγιο σχήματος "L", η γεωμετρία και το μέγεθος του οποίου βρίσκει εφαρμογή στα πτερύγια ψύξης κυρίως τμημάτων υπολογιστή. Μέσω του ανεπτυγμένου κώδικα καθίσταται δυνατός ο έλεγχος περισσότερο πολύπλοκων σχημάτων και διαφόρων υλικών, που συναντώνται στην πράξη. Έπειτα η ροή ενός αραιοποιημένου αερίου μελετάται μεταξύ παράλληλων πλακών απείρου μήκους (μονοδιάστατη ροή) και εντός αγωγού ορθογωνικής διατομής (δισδιάστατη ροή). Η συγκεκριμένη ροή είναι γνωστή στη βιβλιογραφία ως ροή Poiseuille.

Τα αποτελέσματα που προέκυψαν υπέδειξαν υψηλή απόδοση και επιτάχυνση όλων των παράλληλων εκτελέσεων σε σύγκριση με τους αντίστοιχους σειριακούς κώδικες. Συγκεκριμένα, στο πρόβλημα μεταφοράς θερμότητας ο παράλληλος κώδικας προέκυψε μέχρι και 410 φορές γρηγορότερος από τον σειριακό. Αντίστοιχα, στη μονοδιάστατη ροή αραιοποιη-

μένου αερίου επιτεύχθηκε 46 φορές επιταχυνόμενη εκτέλεση, ενώ στη δισδιάστατη ροή έως 298 για διπλή και 568 για απλή ακρίβεια αριθμών. Η ταχύτατη επίλυση με τη χρήση της CUDA επιβεβαίωσε την σπουδαία και εκτεταμένη χρήση της στον επιστημονικό και ερευνητικό τομέα, καθώς και την συνεχή εδραίωση του μοντέλου αυτού.

Contents

Chapters *Page*

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter introductive information is presented, that offers the motivation and the background of this diploma thesis. Initially, general concepts of parallel computing are presented, as well as a literature review on graphical processing units applications. Afterwards, a basic introduction to the theory of rarefied gases takes place, and finally the structure of the diploma thesis is described.

1.1 Parallel Computing

As advances in computing continue to take place the amount of data being processed skyrockets. In many applications, processing the ever-growing data, as fast as it is possible, is the most important factor. Ergo, the tremendous need of accurate and fast results in all scientific fields made the need for parallelism on simulation mandatory.

According to Moore's law, the number of transistors on integrated circuit chips doubles approximately every 18 months [1]. Since Gordon E. Moore has stated this law till now, it is proven to be valid while he also declared that, at least for the short term, this rate can be expected to continue if not to increase. Furthermore, Central Processing Unit (CPU) clock rates do not follow an increasing trend, however they asymptotically tend to an upper limit [2]. The explanation of this behavior is that constantly shrinking the size of transistors to fit inside a CPU is of course limited by physical laws. As a result,

until quantum computers [3, 4, 5] become a reality, additional speed will only be obtained throughout additional cores [6].

Contrariwise to the traditional serial computing, parallel computing is the simultaneous utilization of multiple compute resources to solve a computational problem [7]. The problem is broken into several discrete parts that can be solved independently and thus concurrently. Each one of these parts is further broken down to a series of instructions, which are executed simultaneously on different processors. All these instructions are controlled and coordinated by a mechanism. The compute resources are typically a single computer with multiple processors or an arbitrary number of such computers connected via network. For the time being, supercomputer parallel performance can reach exascale computing levels (1 Exaflop = 10^{18} calculations per second).

Many different parallelism programming models have been developed, such as OpenMP and MPI. The Message Passing Interface (MPI) Standard is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users [8]. The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that is widely used for writing message passing programs [9]. As such, MPI is the first standardized, vendor independent, message passing library. The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility. The MPI interface provides essential virtual topology, synchronization, and communication functionality between a set of processes in a language-independent way, via language-specific syntax and a few language-specific features. MPI is based on Single Instruction Multiple Data (SIMD), which means that a single instruction is executed in parallel on multiple data points as opposed to executing multiple instructions.

OpenMP is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. The API supports C/C++ and Fortran on a wide variety of architectures [10]. OpenMP is thread-oriented in a

Single Instruction, Multiple Thread (SIMT) fashion, meaning that each core is assigned to perform one particular operation per clock cycle.

A parallel model may be more suitable than another for a specific problem. To increase the suitability and peak performance, hybrid models have been developed. A hybrid model is a combination of several parallel programming models in the same program. For example a common hybrid model, that is met in the literature, is the combination of the message passing model (MPI) with the threads model (OpenMP) [11]. In this model threads perform computationally intensive kernels using local, on-node data, while communication between processes on different nodes occurs over the network, using MPI.

Compute Unified Device Architecture (CUDA) is a parallel computing platform and application programming interface (API) model, created by NVIDIA® for general purpose computing on graphical processing units (GPUs). The programmer is capable to develop scripts with parts that are executed in parallel, with minimal alterations to default structure. It makes it possible to perform high speed parallelism, without the need of powerful computer clusters, but only by possessing a CUDA supported GPU.

It is easily perceived that what is necessary, in order to obtain fast results is the utilization of powerful, yet in large numbers cores. However, in many cases where the amount of data being processed are nearly abundant what matters more is not the strength of the cores, but most importantly to have large quantities in possession. Based on this generic idea NVIDIA programmers thought to develop a parallelism model that will run on Graphical Processing Units (GPUs), which are equipped with numerous cores, less powerful from CPUs.

The result of this new programming model, called CUDA™, was quickly and effortlessly adapted by both the industry and the scientific community. Numerical analysis and simulations gained a considerable increase in performance from a single GPU, that in many cases was above even whole CPU clusters. An additional enhancement was introduced with the GPU clusters, offering great computational capabilities; eradicating the boundaries of many time consuming applications.

GPUs were originally designed to perform all necessary calculations required for three-

(a) (b)

Figure 1.1: (a) CPU and (b) GPU transistors. The GPU has a clear advantage for data processing operations.

dimensional graphics. Hence, they are extremely powerful devices as, in order to produce graphic results a lot of calculations, though simple ones, need to be performed for every single frame [12]. If one considers a usual GPU with the capability to provide 60 frames per second the resulting necessary calculations may be at the magnitude of millions at every second. As seen in Figure 1.1, in a GPU most transistors are dedicated to process data rather than caching and controlling the flow.

During the past few years, since its introduction in November 2006, CUDA has witnessed a tremendous growth. In the field of software development, computational fluid dynamics and product simulations the need of parallel multiprocessing is already mandatory, in order to withstand the tremendous data processing load. In addition the broad necessity for advanced computational power drives programmers in alternative solutions to immense and rarely found, powerful computer cluster structures of research centers and big companies. With CUDA it is possible to achieve high computational bandwidth using one or more multi-core GPU, on ones' personal computer.

The performance of GPU cores is usually severely worse than the one of CPU cores. Though where GPUs have a clear advantage is at the excessive number of integrated cores

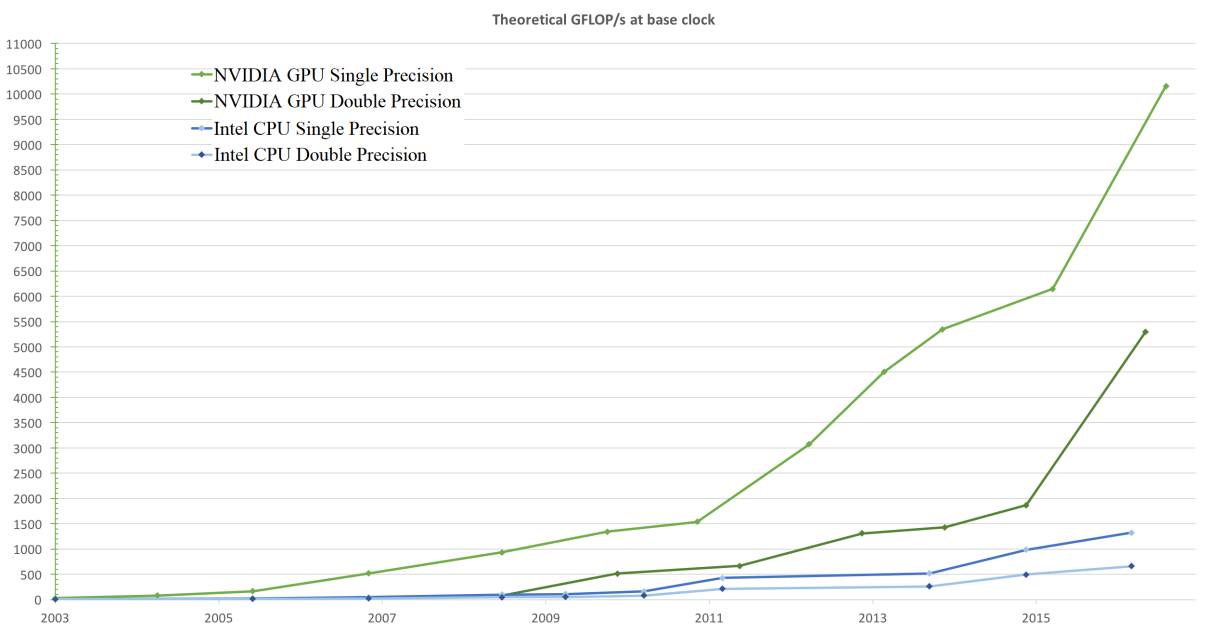


Figure 1.2: Theoretical GFLOP/s at base clock of NVIDIA®GPUs and Intel CPUs [2].

per each processor in comparison to the small collection of powerful cores a CPU has, as they were designed to perform intensive, highly parallel computations for graphics. At the time of writing a usual highly powerful CPUs has around 8 to 16 cores, whereas a GPU has thousands weaker –in means of clock speed (GHz)– cores. The overall result is the GPUs to achieve profound GFLOP/s, an essential metric of performance as discussed later on. In Figure 1.2 one can observe the great improvement of GPU performance, over the years, in relation to the reluctant advancement on CPU technology.

1.2 Graphical Processing Unit applications

Since its debut in early 2007, parallel computing on GPUs has already been utilized by the scientific community, offering great benefits, often by orders-of-magnitude performance improvement compared to the previous state-of-the-art implementations [13].

Using CUDA API, programmers could offer a huge assist towards many scientific fields that greatly suffered from computations. For example, CUDA helped accelerating

cancer detection. An extremely efficient and rapid way to early detect breast cancer, is based on three-dimensional, ultrasound imaging method, but its solution had not been put into practice due to computational limitations. TechniScan Medical Systems utilizing the power of GPUs made possible to use ultrasonic waves to image the patient's chest and within 20 minutes the doctor can manipulate a highly detailed, three-dimensional image of the woman's breast and make a diagnosis [13]. This tool offered a huge help to humanity and saved lots of peoples lives.

The contribution of CUDA to the mechanical engineering field is not less significant. For many years, the design of highly efficient rotors and blades remained a black art of sorts. The astonishingly complex movement of air and fluids around these devices cannot be effectively modeled by simple formulations, so accurate simulations prove far too computationally expensive to be realistic. Only the largest supercomputers in the world could hope to offer computational resources on par with the sophisticated numerical models required to develop and validate designs. Since few have access to such machines, innovation in that direction continued to stagnate [13]. Many publications have been made towards the implementation of CUDA for rotors and airfoils, such as the study of [14], which presents the porting of two- and three-dimensional Navier-Stokes equations solvers for unstructured grids, or the study of [15], in which the implementation of two separate compressible flow solvers is presented.

GPUs offered a huge assistance, even for more demanding and time consuming mechanical engineering problems. A strategy for implementing CUDA on solvers for partial differential equations (PDEs), that rely heavily on stencil computations on three-dimensional, multiblock structured grids is presented in [16]. These solvers find application in the field of computational fluid dynamics (CFD) for flows in turbomachines, such as jet engines and land-based gas turbines. The motivation for that research study was to run the solvers in parallel, on computers with multiple GPUs. The performance of the parallel code was satisfactory, mainly for large numbers of utilized GPUs.

Furthermore, developed parallel codes in the field of rarefied gases showed enormous accelerations [17]. In this research study an algorithm specifically for solving model kinetic

equations onto GPUs was accomplished. The efficiency of the algorithm is demonstrated by solving the one-dimensional shock wave structure problem and the two-dimensional low Mach number driven cavity flow, showing that it is possible to cut down the computing time of the serial codes by two orders of magnitude.

CUDA API was integrated into many common commercially available software packages, to speed up their execution time. ANSYS® Fluent® accelerated a linear equation solver, as well as models for radiation heat transfer and discrete ordinate radiation, offering up to 3.7 times faster execution on GPUs [18]. ANSYS® Mechanical® is a finite element analysis tool for structural analysis, including linear, nonlinear and dynamic studies, which introduced GPU support for various solvers. Additionally MATLAB® turned towards CUDA, in order to speed-up highly time consuming tasks, such as Artificial Intelligence (AI), deep learning, and other computationally intensive analytics [19].

Moreover, many research studies in various fields have been carried out using CUDA Fortran, as in physics [20], atmospheric climate [21], computer science [22], turbomachinery [23], mathematics [24] and others. Ergo, CUDA is a very efficient and popular tool, that helps to accelerate heavy work load programs in different and diverse scenarios.

1.3 Basic concepts of kinetic theory of gases

Rarefied gas dynamics (RGD) is the study of phenomena taking place at an arbitrary ratio of the mean free path (or time) to the characteristic dimension (or time) of the phenomena. RGD an adequately explored field, but still with abundant undiscovered land, greatly suffers from extreme computational load. Since the developed simulations take place in a microscopic level, meaning in terms of properties of the individual molecules (inter-molecular force law), in many cases the serial execution time greatly exceeds bearable duration. Ergo, the utilization of parallel techniques proves not only beneficial, but in many cases mandatory to obtain a swift solution. CUDA API is an exceptional alternative, that with basic knowledge of only the fundamentals the researcher can obtain results extremely faster in comparison to the serial code, but also to other alternative

parallelism techniques. Its greatest advantage is probably that tremendous performance can be granted from an ordinary NVIDIA GPU of a personal computer, without the need of huge clusters.

The simulation of rarefied gas flows is of high importance in many fields including high altitude aerodynamics [25, 26], vacuum technology [27, 28] and Micro Electronic Mechanical Systems (MEMS) industry [29, 30]. In the field of aerospace great attention is paid in the development of micro-propulsion systems, such as mono- and bi-propellant thrusters, for satellites [31], as well as reentry of orbiting vehicles [32]. In vacuum technology that deals with the development of equipment operating under near or ultra-high vacuum conditions depends greatly on rarefied gas dynamics theory and applications [33]. Moreover, the design and optimization of MEMS in sizes from few millimeters down to micrometers is currently an emerging field with much scientific attention and an immense need for accurate simulation of rarefied gas flows [34].

In classical fluid mechanics the fluid is considered as a continuous medium, where the hydrodynamic equations can be successfully applied. Although in order for a fluid to be characterized as continuous certain criteria must be met. The first is that the characteristic size of gas flow must be considerably larger than the molecular mean free path (MFP), which is the distance a particle of a gas travels between two successive intermolecular collisions [35]. Secondly, for nonstationary flows the mean free time of gaseous particles, which is the time between two successive collisions, must be significantly smaller than a time interval, in which a change occurs at a macroscopic variable. Many cases exist where both or one of two of the above mentioned assumptions are not fulfilled, hence the use of a model at a microscopic model is mandatory.

The gas is classified as rarefied according to the *Knudsen number*. The Knudsen number is a dimensionless number defined as the ratio of the molecular mean free path (λ) to a characteristic size of gas flow or a length scale of macroscopic gradient ($D = \rho/\partial\rho/\partial x$, where ρ is the fluid density), given by Equation 1.1 [36],

$$Kn = \frac{\lambda}{D}. \tag{1.1}$$

According to the hard sphere (HS) model, where molecules are considered as spheres, the mean free path can be written as:

$$\lambda = \frac{8}{5}\sqrt{\frac{2k_BT}{\pi m}}\frac{\mu}{p}, \tag{1.2}$$

with μ being the dynamic viscosity of the gas in temperature T, p is the pressure, m is the molecular mass and k_B is the Boltzmann constant equal to $1.380649 \times 10^{-23} J/K$ [37].

Another dimensionless number that is usually met in the literature is the rarefaction parameter δ, because it can be computed according to macroscopic quantities.

$$\delta = \frac{D}{l} = \frac{pD}{\mu u_m} = \frac{\sqrt{\pi}}{2}\frac{1}{Kn}, \tag{1.3}$$

where l is the equivalent free path and u_m is the most probable speed. The rarefaction parameter is inversely proportional to the Knudsen number as the equivalent free path is proportional to the mean free path. Also the most probable speed is defined as:

$$u_m = \sqrt{\frac{2k_BT}{m}}. \tag{1.4}$$

These two dimensionless numbers define the four rarefaction regions. These regimes are the hydrodynamic, slip, transitional and free molecular regime. It is important to note that the limits of these regimes are not robust, however the most acceptable ones are given below.

- For $Kn < 10^{-3}$ or $\delta > 1000$ the *Hydrodynamic regime*. Fluid can be characterized as a continuum medium and the Navier-Stokes equations can be applied.
- For $10^{-3} < Kn < 10^{-1}$ or $1000 > \delta > 100$ the *Slip regime*. Non-equilibrium begins to be substantial in the boundaries of the domain, with velocity slip and temperature

jump phenomena, i.e. the gas and the walls do not have the same velocity or temperature. The continuum model is still valid, if certain modifications of the boundary conditions at the solid walls are applied.

- For $10^{-1} < Kn < 100$ or $10 > \delta > 10^{-2}$ the *Transitional regime*. Intermolecular collisions are reduced substantially and the distribution is not of Maxwellian type, thus a kinetic description of the gas becomes essential.
- For $Kn > 100$ or $\delta < 10^{-2}$ the *Free molecular or regime*. Its the regime where molecules remain unaffected by each other, thus no intermolecular collisions occur.

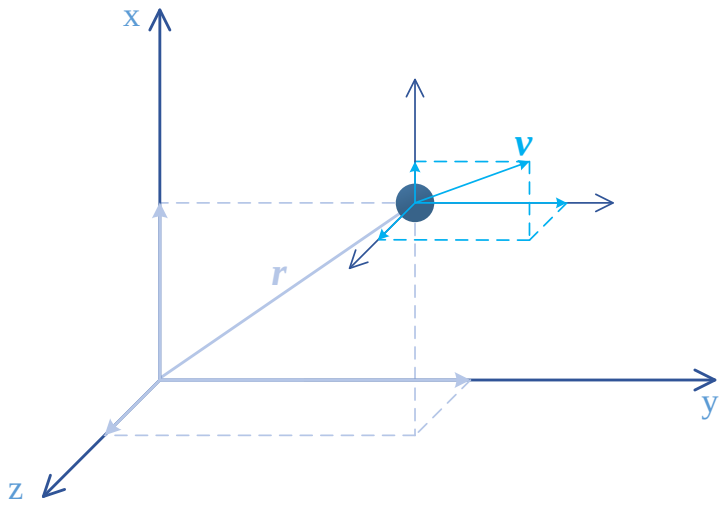


Figure 1.3: Physical and molecular velocity spaces of a particle.

Ludwig Eduard Boltzmann (1844 - 1906) was an Austrian physicist famous for his research on statistical mechanics in order to study the behavior of molecules in a gas [38]. The equation he derived (Equation 1.5), known as *Boltzmann equation*, describes the dynamics of an ideal gas [35]. The most important assumptions made to derive Equation 1.5 are the existence of only binary collisions, which is valid for gases at low densities, and the hypothesis of *molecular chaos*, meaning that the colliding particles' velocities are uncorrelated and statistically independent of their position.

$$\frac{\partial f}{\partial t} + \boldsymbol{v} \cdot \frac{\partial f}{\partial \boldsymbol{r}} + \boldsymbol{F} \cdot \frac{\partial f}{\partial \boldsymbol{r}} = Q(f, f'), \tag{1.5}$$

where f is the velocity distribution function, vector $\boldsymbol{v}$ denotes the velocity vector of the particle and r denotes the position vector of the particle, as shown in Figure 1.3. The first term of the left hand side of Equation 1.5 refers to time variation, the second to spatial variation and the third to the effect of the force F acting on the particle. The term on the right hand side $Q(f, f')$ denotes the integral describing binary intermolecular collisions. The collision operator is defined as:

$$Q(f, f') = \int\int\int \left(f' f'_* - f f_*\right) g b db d\epsilon d\boldsymbol{v}_*, \tag{1.6}$$

where, f, f_* are the pre-collision distribution functions of the two particles and f', f'_* are the post-collision ones. Moreover, $g = |\boldsymbol{v} - \boldsymbol{v}_*|$ is the relative velocity, b is the impact parameter and ϵ is the azimuthal angle. These parameters for the hard sphere model is shown in Figure 1.4.

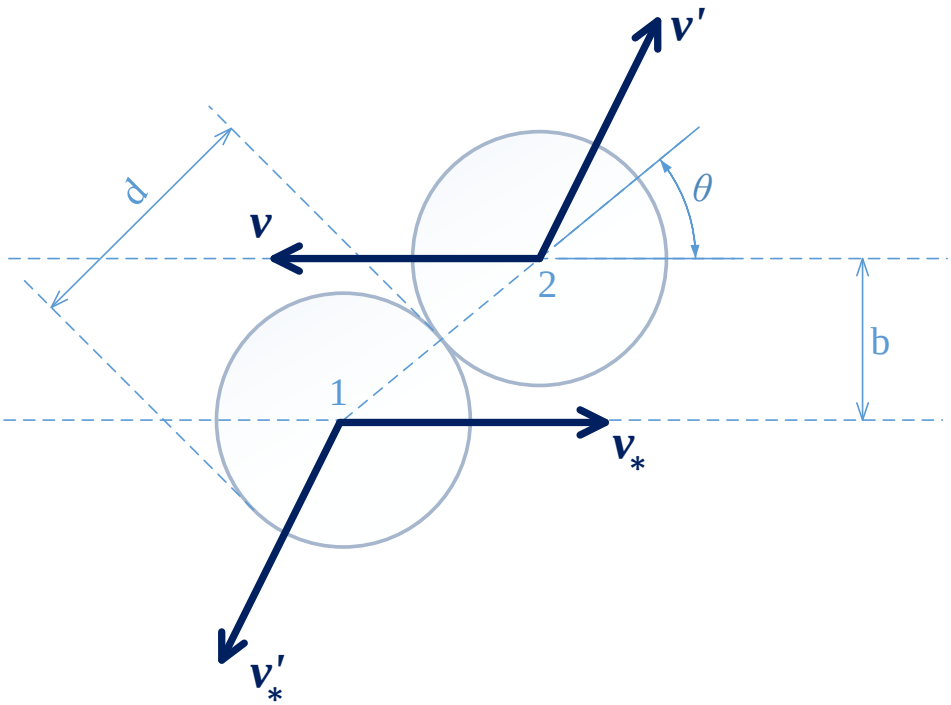


Figure 1.4: Collision of hard sphere model particles.

Solving the Boltzmann equation for the unknown distribution function leads to the

determination of the gas macroscopic quantities, such as the following:

- Number density

$$n(\boldsymbol{r},t) = \int_{-\infty}^{\infty} f d\boldsymbol{v} \tag{1.7}$$

- Gas bulk velocity vector

$$\boldsymbol{u}(\boldsymbol{r},t) = \frac{1}{n(\boldsymbol{r},t)} \int_{-\infty}^{\infty} \boldsymbol{v} f d\boldsymbol{v} \tag{1.8}$$

- Pressure

$$P(\boldsymbol{r},t) = \frac{m}{3} \int_{-\infty}^{\infty} (\boldsymbol{v}-\boldsymbol{u})^2 f d\boldsymbol{v} \tag{1.9}$$

- Stress tensor

$$\hat{P}_i j(\boldsymbol{r},t) = m \int_{-\infty}^{\infty} (v_i - u_i)(v_j - u_j) f d\boldsymbol{v} \tag{1.10}$$

- Temperature

$$T(\boldsymbol{r},t) = \frac{m}{3k_B n(\boldsymbol{r},t)} \int_{-\infty}^{\infty} (\boldsymbol{v}-\boldsymbol{u})^2 f d\boldsymbol{v} \tag{1.11}$$

- Heat flux vector

$$Q(\boldsymbol{r},t) = \frac{m}{2}\int_{-\infty}^{\infty}(\boldsymbol{v}-\boldsymbol{u})^2(\boldsymbol{v}-\boldsymbol{u})f d\boldsymbol{v} \tag{1.12}$$

From equations 1.9 and 1.11 it is proven that the ideal law of gas is valid even at non-equilibrium state, and is described by:

$$P(\boldsymbol{r},t) = n(\boldsymbol{r},t)k_B T(\boldsymbol{r},t). \tag{1.13}$$

Obtaining an exact numerical solution of the Boltzmann Equation 1.5 with the exact collision integral 1.6 is extremely time consuming even by using powerful computers. In order to overcome this difficulty, simplified equations that satisfy the main properties of the Boltzmann equation have been suggested. The BGK model proposed by [39] and independently by [40] was the first kinetic model proposed and was widely used till then, mostly due to its simplicity, among others, such as the Shakhov model [41], the Ellipsoidal model [42] etc. According to this model, the collision part of the Boltzmann equation is substituted by:

$$Q\left(f, f^M\right) = v_B\left(f^M - f\right), \tag{1.14}$$

where f^M is the local Maxwellian, which contains local values of the number density $n(t,\boldsymbol{r})$, bulk or macroscopic velocity $u(t,\boldsymbol{r})$ and temperature $T(t,\boldsymbol{r})$, and v_B is the intermolecular collision frequency, assumed to be independent of molecular velocity. The quantities $n(t,\boldsymbol{r})$, $u(t,\boldsymbol{r})$ and $T(t,\boldsymbol{r})$ contained by f^M are unknown.

The BGK model gives quite satisfactory results in the whole range of the Knudsen number. However it has certain limitations. Specifically, the collision frequency must be

adjusted every time according to the flow requirements and one cannot obtain both viscosity and heat conductivity transport coefficients simultaneously and correctly, resulting in a Prandtl number of unity for monoatomic gases, instead of the correct value 2/3.

It is extremely difficult to solve the Boltzmann transport Equation 1.5, due to the seven dimensions of the distribution function and the collision integral term. From the numerical methods introduced in the literature the most widely used deterministic methods is the Discrete Velocity Method (DVM) [43]. According to DVM a discrete set of molecular velocities is considered when integrating the distribution function during the calculation of the macroscopic quantities. Hence, the kinetic equations in the physical and molecular velocity space can be solved only for these discrete velocities and obtain a solution through an iterative method. Using many discrete velocities will guarantee more accurate results. To make the optimal choice of velocities, which will lead to more accurate results of the macroscopic properties of the fluid, the roots of an orthogonal polynomial will be administered, in order to apply a numerical integration, such as Gauss, Legendre, Trapezoidal rule, Chebyshev etc. The iterative method leads to convergence in general, however the necessary number of iterations becomes significant as Knudsen number becomes smaller, i.e. as the flow gets closer to the hydrodynamic regime.

Another way is to simulate directly all molecules in a probabilistic manner, considering all their collisions and applying the laws of motion. This method is called Direct Simulation Monte Carlo (DSMC) and is of statistical nature [44]. The DSMC gives very accurate results in the whole range of Knudsen number. However, there are cases where this model greatly suffers. For example, for the studied case with a small pressure difference between the vessels or for a Couette flow with a very small velocity of the plate there may be observed great oscillations in the results. In this area the linear kinetic theory is valid and is applied to obtain reasonable results. Moreover, DSMC simulations for large values of rarefaction parameter δ require great computational power and time to obtain a solution, making it impractical for many cases [45].

1.4 Diploma thesis scope and structure

The objective of this diploma thesis was to study benchmark mechanical engineering problems on GPUs, which leads to a considerable reduction of the required simulation time. The implementation of parallelism techniques on the developed serial code was accomplished via the CUDA application programming interface, developed by NVIDIA.

The rest of this diploma thesis is separated into three chapters, 2-5:

- Chapter 2: CUDA Fortran Application Programming Interface is described analytically. Specifically, a thorough examination of the programming and hardware model is presented, as well as the most important performance bottlenecks and various ways to eliminate them.

- Chapter 3: The heat transfer through an L-shaped fin is examined along with the acceleration of the code using CUDA. The studied geometry and dimensions along with the material of the fin is mainly used for cooling computer parts, such as CPU processors. The fin is subjected to convection and heat flux from the boundaries to simulate the exact conditions met on cooling fins.

- Chapter 4: The implementation of CUDA on an one-dimensional flow of a rarefied gas between two parallel plates of infinite width is studied using kinetic equations. The flow occurs due to a slight pressure difference among both edges, also known as Poiseuille flow. The temperature is held constant throughout the domain.

- Chapter 5: The two-dimensional flow of a rarefied gas through a rectangular cross section duct is investigated. The duct connects two vessels with the gas at different pressure and same temperatures. Due to the small pressure gradient the gas is subjected to a laminar flow, or Poiseuille flow. This problem is proven extremely computationally demanding, thus CUDA parallelism appears highly beneficial.

Chapter 2

Basic aspects in CUDA Fortran

At the beginning, CUDA was adapted to C programming language, but currently it has been extended to other languages as well (like C++, Fortran, Java, Python etc.). CUDA Fortran, developed jointly by Nvidia® and Portland Group® Inc. (PGI) [46], is an alternative to the primary presented CUDA C, that offers the capability to develop software using Fortran programming language. In the fields of research Fortran is still preferred, as it presents some advantages on execution speed and ease of optimization in regard to other languages. In this diploma thesis the choice of using exclusively Fortran was made.

2.1 Programming Model

In the specter of parallel computing it is essential to clarify some important terms. The CPU and its memories are called the *host*, whereas the GPU and its memories are called the *device*. A subroutine that is executed exclusively on a device is called a *kernel*. A program can never succeed 100% parallelism as it will always include parts, that will run serially. In Figure 2.1 a schematic representation of a program is shown, including the parallel and serial code segments.

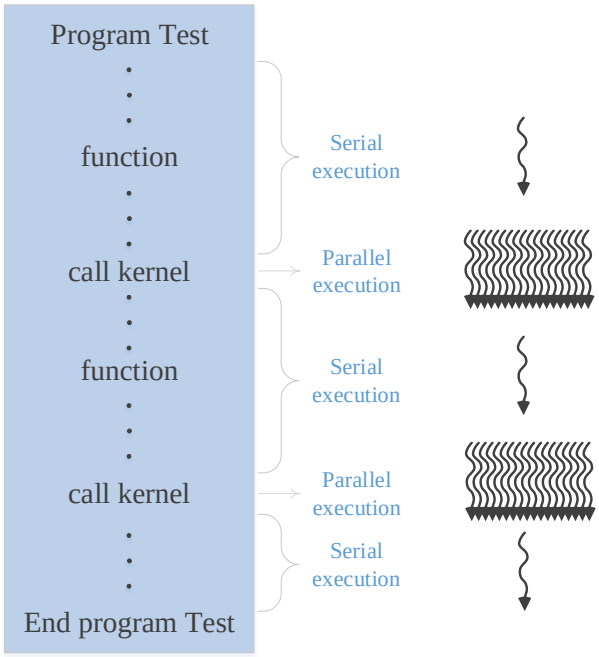


Figure 2.1: A schematic representation of serial and parallel executions of a program.

2.1.1 Thread

CUDA is based on the simultaneous parallel execution of multiple processes called *threads*, thus performing what is called *thread-level parallelism* [47]. In computer science, executing a thread means that a processor executes a sequence of instructions [48]. Hence, thread-level parallelism is when the software is able to specify multiple threads to be executed and the hardware is capable of executing these threads concurrently. Thread-level parallelism is the alternative to instruction-level parallelism, which is when a number of independent operations is issued by a single thread. Since some threads may run faster than others, the instruction is carried out randomly by the threads during execution, mainly based on the load of the processor. Therefore, the programmer should not rely on calculations that are based on consecutive thread order, as all threads begin together, but some finish faster than others.

Inside a kernel, the thread's identity is defined by the default CUDA structure, of Fortran language, as `threadIdx`, which is a 3-component vector for every Cartesian coordinate. Hence, `threadIdx%x` corresponds to x direction and `threadIdx%y`, `threadIdx%z`

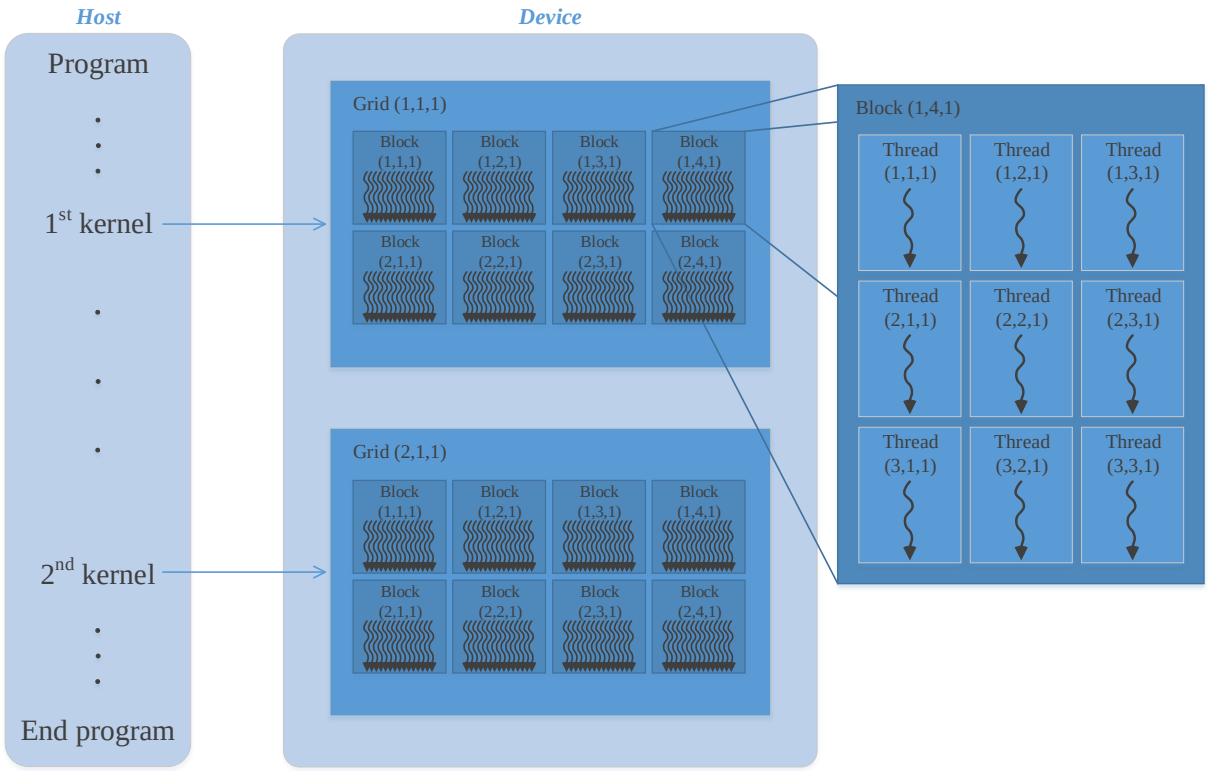


Figure 2.2: A schematic representation of CUDA programming model. A kernel is invoked by a grid of thread blocks each one of which consists of multiple threads.

to y and z direction respectively. A `threadIdx%` with the suffix `x`, `y` or `z` will give back a number from 1 till the total number of threads per thread block in the respective coordinate x, y, z. A thread's identity does not alter during execution of a single kernel, as it is defined at launch and remains the same, till completion is succeeded.

In case it is vital to synchronize all threads of a block, for example to collect data from shared memory (see subsection 2.3.4), in order to prevent performance bottlenecks, like bank conflicts (see subsection 2.5.3), one can use the command `call syncthreads()` inside the kernel, which will result in every thread to pause and wait till all other ones reach the exact same point where this function is called. The command `call syncthreads()` is expected to be lightweight, thus negligibly affects kernel execution time [2].

2.1.2 Thread block

Subsequently, threads are grouped into *thread blocks*, to counterbalance better processing and data mapping. Every thread block utilizes different resources and accesses certain memories autonomously, as later on explained. All threads of a single block can communicate with each other but not with threads of different blocks. A block can either be one-, two- or three-dimensional, a property that is specified at its initial declaration on host.

There are two parameters that define a thread block and can be used inside a kernel. The first one is, similar to threads, the block's identity, which is defined, by the CUDA structure, as a 3-component vector `blockIdx%`. The suffix `x`, in `blockIdx%x`, represents the x direction and can also be `y` or `z` for the y and z coordinate respectively. Moreover, another important parameter is the block's dimension `blockDim%x` in x direction or `blockDim%y` and `blockDim%z` for the other two. The block dimension parameter, as stated by the name, gives the number of threads per block and is extremely useful for indices inside a kernel.

On host the block variable name can either be declared as integer if it is one-dimensional or with the attribute `type(dim3) ::` followed by the variable. The CUDA API's type `dim3` gives the ability to the programmer to declare two- or three-dimensional blocks. Before the kernel is launched, the dimensions of a block, for example with the name `tBlock`, can be initialized as `tBlock = dim3(16,16,1)`. As a consequence, the kernel will be launched by blocks of 16 threads in the x- and y-direction and 1 in z, thusly by two-dimensional blocks.

The amount of threads per block is limited and depend on the device one has in possession. On devices of compute capability 1.3 or less the total number is restricted to 256, whereas for higher compute capability the upper limit is 1024. Additionally, a kernel can be launched with a maximum of $2^{31} - 1$ blocks for compute capability of 3.0 and higher [2].

2.1.3 Grid

Lastly, many thread blocks consist a single *grid*. The grid can also be declared as one-, two- or three-dimensional. In Figure 2.2 the programming model of CUDA is presented, with a kernel being invoked by a grid of various threads and thread blocks.

The programmer is able to manipulate the number of threads contained in every block and the number of blocks per grid (see section 2.4.1), each one of which has a unique address. When accessing a specific dimension of an array inside a kernel, the index must be valid throughout all different threads and thread blocks. If the dimension of the array is small enough to be called with less than 1024 elements (not to exceed the devices thread limits), then the kernel can be called with one block and 1024 threads without any difficulties. In that case, using an `integer` variable `index`, the whole matrix can be accessed without any problems using the index: `index = threadIdx%x`. Complications arise when the kernel is called with several thread blocks. As previously explained a thread identity is unique only for each block, hence a correlation must be made between threads and blocks.

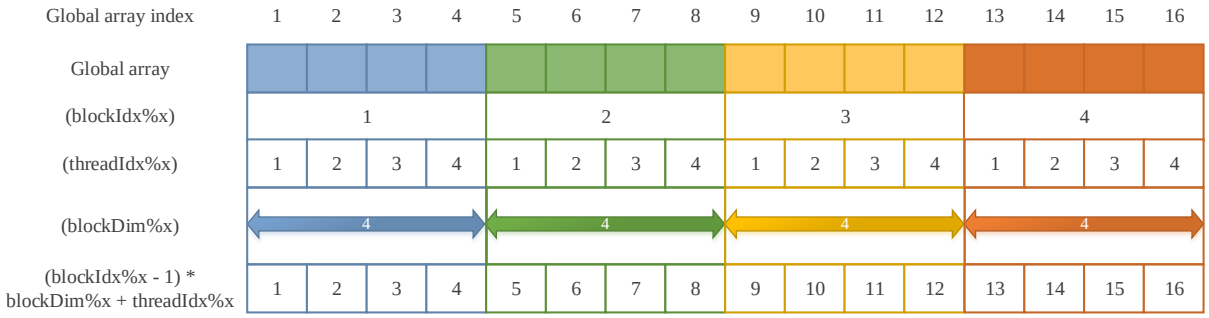


Figure 2.3: A schematic representation of device hardware.

Using the aforementioned CUDA structure variables, `threadIdx%x` and `blockIdx%x`, the global array index is accurate when defined as `(blockIdx%x - 1) * blockDim%x + threadIdx%x`. The minus one in the latter formula is placed, because in Fortran language all indexes start from one and not zero. In Figure 2.3 the logic to derive the global array index is schematically presented. Specifically, what it must be accomplished is the global array index to take all values from 1 to 16 exactly once. There are four blocks of size four,

thus the `blockIdx%x` takes values from 1 to 4, whereas the block's dimension `blockDim%x` is equal to 4. Additionally, there are four threads per block with identities 1–4. Applying the aforementioned index it is guaranteed that the accesses to the global array will have the desired sequence.

A kernel is called with one grid, thus one should be careful not to exceed the device's capabilities, in the means of maximum threads per block and maximum thread blocks. In case the total number of threads per grid surpasses the limit the programmer can make use of a technique called *grid-stride loops*, that increases kernel flexibility [49]. According to this technique the number of elements is divided to the number of threads per grid, in order to obtain the total number the kernel should be launched. Outside the kernel, on host, one *do-loop* guarantees that the kernel will be launched various times, in order to cover the whole range of elements. The step of the do-loop is fitted with the grid dimension and is incremented by `blockDim%x * gridDim%x` at every iteration. In Table 2.1 the programming model characteristics of the different GPUs used for this diploma thesis are presented.

Name	**maximum threads per block**	**maximum block dimensions**	**maximum grid dimensions**
NVidia Quadro M2200	1024	$1024 \times 1024 \times 1024$	$2147483647 \times 65535 \times 65535$
NVidia Titan Xp	1024	$1024 \times 1024 \times 1024$	$2147483647 \times 65535 \times 65535$

Table 2.1: CUDA features of the GPUs that were used.

2.2 Hardware Architecture

The way CUDA programming model runs on device is different than the way one programs it. Despite the fact that the programmer can only alter the number of threads and thread blocks, in order to peak performance and eliminate performance bottlenecks, one should also be familiar with the "hardware model" of CUDA.

2.2.1 Warp

Threads of a block are executed on an SP in groups called *warps*. A warp consists of 32 threads. For devices of compute capability less than 2.0, on an SP only half warps could run concurrently, while on newer versions the whole warp is executed. All threads of a warp run in a single-instruction, multiple-thread (SIMT) fashion, meaning simultaneously and extremely fast and carry out one single instruction. That is done in case no branching exists, creating thread divergence (see subsection 2.5.2 for more details). To accomplish optimum performance, it is wise to run a kernel with number of threads per block, which is a multiple of that number.

The total number of warps in a block is shown in Equation 2.1, where t is the number of threads per block, 32 is the warp size and $ceil(x)$ is a function that gives the smallest integer greater or equal to x;

$$ceil\left(\frac{t}{32}\right). \tag{2.1}$$

2.2.2 Streaming Processor and Streaming Multiprocessor

The basic computational unit on a GPU is the *thread processor*, which is also known as *streaming processor* (SP) or simply as CUDA *core*. A thread processor or core is a floating-point unit. Processors are subsequently grouped into *streaming multiprocessors* (SM or SMP). When a kernel is launched from the host, the blocks of the grid are enumerated and subsequently distributed to SMs with available execution capacity. It is the SM's duty to create, manage, schedule, and execute threads in warps. SMs are designed to execute hundreds of threads concurrently in warps, in single-instruction, multiple-thread (SIMT) fashion, meaning that every thread carries out a single instruction [2]. If a warp does not execute a single instruction and the execution path is different for some threads, then thread divergence occurs, which forces serial thread execution of a warp on an SP, an important performance bottleneck especially as the number of threads increases (see

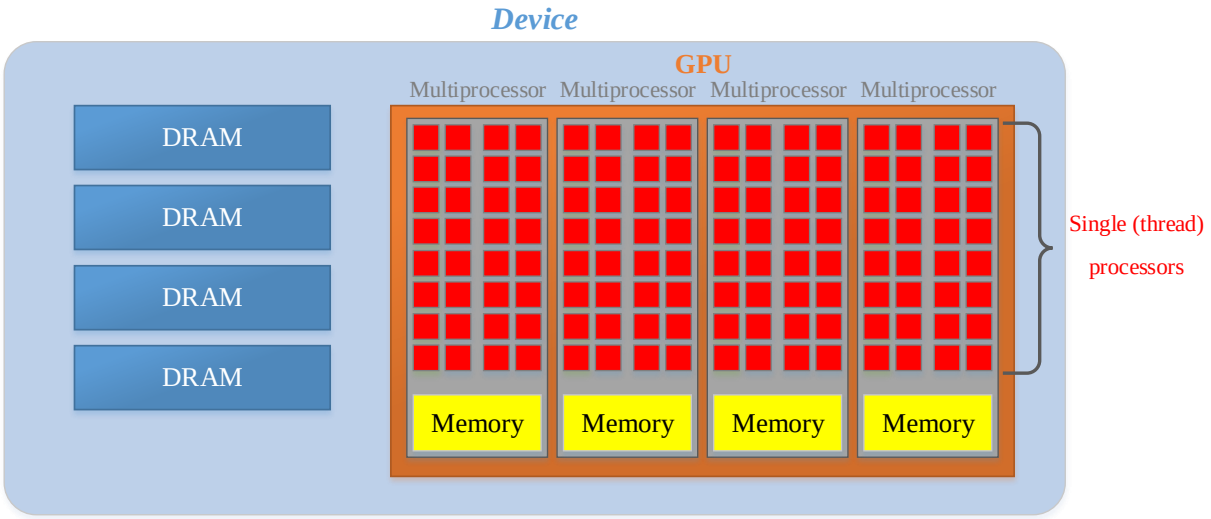


Figure 2.4: A schematic representation of device hardware.

section 2.5.2).

A multiprocessor has access to a set of 32-bit registers and a parallel data cache or shared memory; all private resources that are used by resident threads. As presented in Figure 2.4 a CUDA device is a GPU with several SMs, each one of which contains many SPs. Outside, the *warp scheduler* decides when and which threads will run on each CUDA core. The programmer cannot affect this process. The threads of the warp that execute the current instruction are called *active* threads, whereas the ones waiting are called *inactive*. In tables 2.2 and 2.3 the hardware characteristics of the two GPUs and the CPU used for this diploma thesis are presented.

Name	**Architecture**	**Max Clock Rate**	**Cores per SM**	**SMs**
NVidia Quadro M2200	Maxwell	1.036 GHz	128	8
NVidia Titan Xp	Pascal	1.582 GHz	128	30

Table 2.2: Hardware features of the GPUs that were used.

Name	**Max Clock Rate**
Intel® Core™ i7-7820HQ	3.9 GHz

Table 2.3: Hardware features of the CPU that was used.

2.3 Device Memories

In order to parallel execute a kernel on a device, all involved data in the calculations must be copied there. That requires device memory allocation and subsequently deallocation, along with at least two data transfers – one for data needed by device to perform the necessary computations and one for data results needed back to host–. One should think the GPU as an autonomous system for as long as the calculations take place. The device is equipped with several memories, the main of which has large capacity, at a magnitude of several gigabytes, to successfully store large data files. Following the growth of technology, the memories acquire more and more storage capacity, as well as improving considerably the transferring speed. Another important breakthrough is the introduction of unified memory on GPUs of compute capability of 6.0, that eliminates the need for any data transferring, as there only exists one simultaneously managed memory, accessible by both the host and device.

In Figure 2.5 the different memories that are integrated on a GPU and the way device and host communicate with them is shown with arrows. A line with arrows on both ends indicates that the memory is readable and writable, whereas if a single arrow exists then it is either read only or write only memory. Moreover, in Figure 2.5 the memories accessible by threads, thread blocks or grid are presented. Correct utilization of memories inside a kernel is vital for high optimization.

2.3.1 Global Memory

The main device memory is called *global memory*. Global memory is accessible and writable by both the host and device and is used to store all data that one or several kernels will exploit. Hence, all threads and thread blocks of all kernels have access to the same stored data in this memory, that will remain not only for as long as they run but even after their completion. After all necessary operations are finished, by one or multiple kernels, the needed data results must be copied back to the host. As aforementioned, this memory has the capacity of gigabytes, with advanced GPUs used specifically for

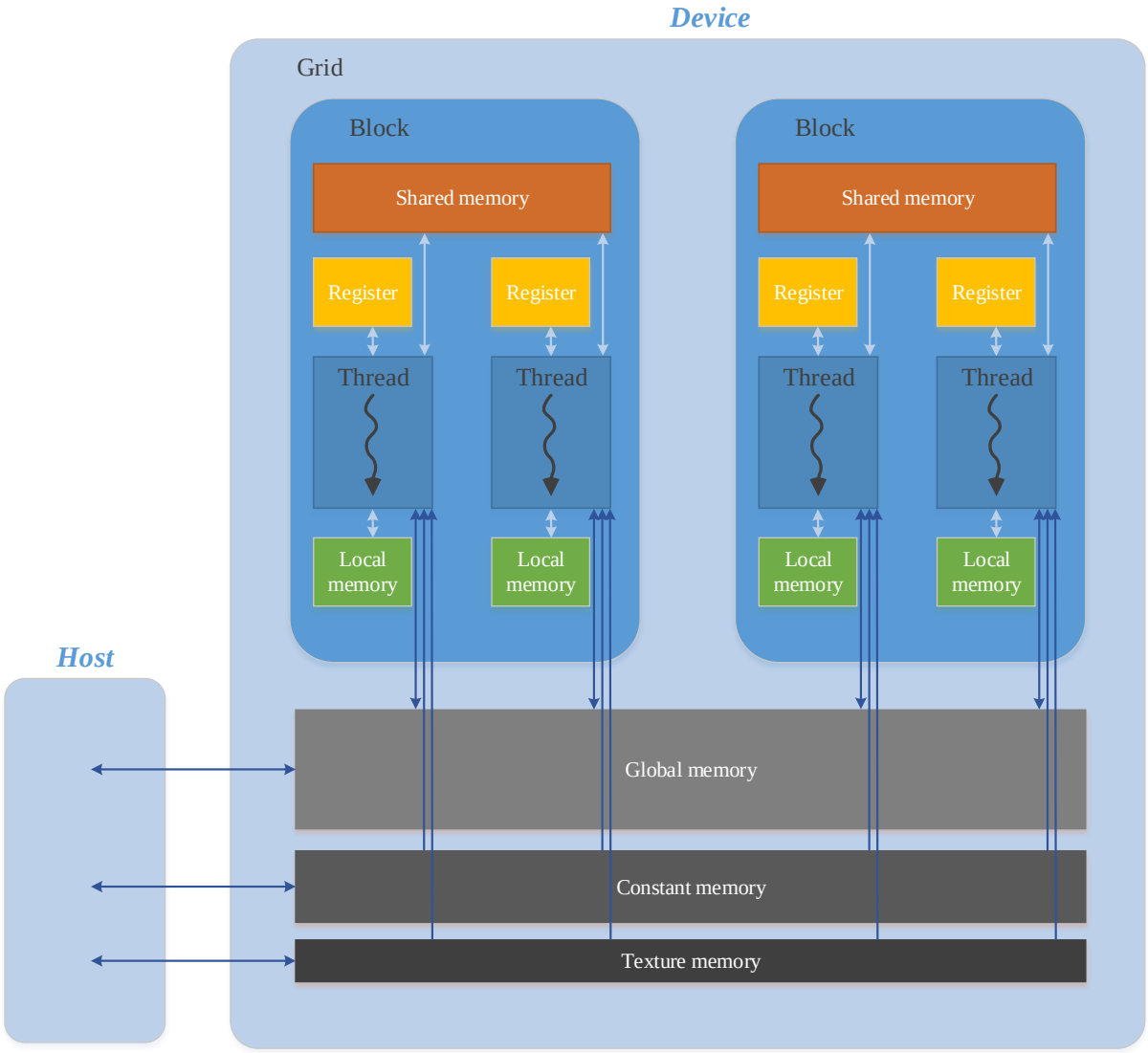


Figure 2.5: A schematic representation of device memories.

programming usually having more than a dozen. For example the two GPUs utilized for this diploma thesis are the NVIDIA Quadro M2200 and NVIDIA Titan Xp that have global memory capacity of 4 and 12 GB respectively. Apart from its advanced storage limits, global memory is a very slow memory –specifically slowest of every other device memory–, having a fairly low bandwidth and is one of the main reasons that a kernel may be sluggish during execution. The utilization of other device memories is crucial to peak performance.

2.3.2 Constant Memory

Another important memory that is accessible by all block of threads is the *constant memory*. It is a "read only" memory by device, meaning that although the host can both write and read, the device is able to obtain data but not write. It is useful for storing data that will not change during kernel execution. In this case it is highly preferred in comparison to global memory, as it has short latency and higher bandwidth when all threads of an active warp simultaneously access the same location.

Careful handling must be done during accessing this memory, because in case of bank conflicts –a term that belongs to one of parallelism bottlenecks– the accesses become serialized, downgrading kernel's performance. Particularly, if threads of a half warp, for compute capability < 2.0, or of a warp, for compute capability > 2.0, access different locations of memory banks, the execution becomes serialized and thus the theoretical high bandwidth collapses.

2.3.3 Texture Memory

Another read-only memory is the *texture memory*. Texture memory is in reality global memory, that is accessed through a dedicated read-only cache [50]. As NVIDIA designed the texture memories for the OpenGL® and DirectX® rendering pipelines, they are much better in comparison to constant memory when accessed in a way that greatly depends on spatial locality. As seen in Figure 2.6, although the accesses are not consecutive and would not be cached together using other memories, texture cache is designed in a way to hasten similar accesses. As a result, texture memory is helpful and beneficial when a kernel accesses uncoalesced memory banks, a serious performance bottleneck that is described in Section 2.5. CUDA programmers exploit this memory usually when dealing with uncoalesced accesses to global memory or for constant arrays.

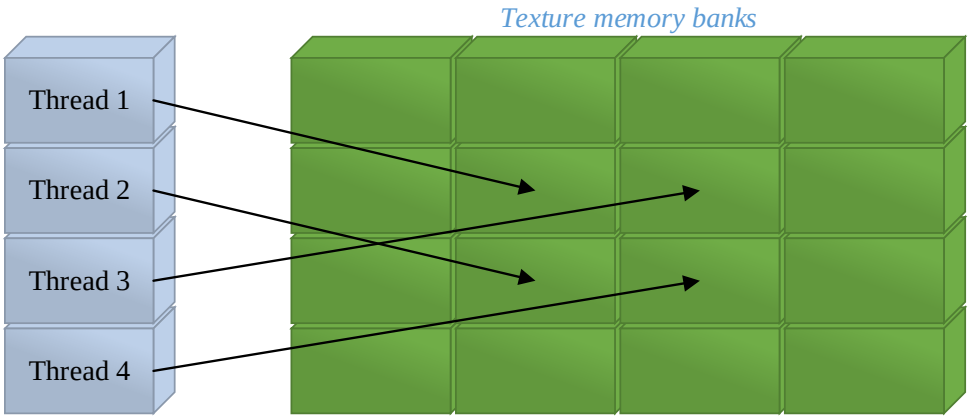


Figure 2.6: A schematic representation of threads accessing texture memory banks.

2.3.4 Shared Memory

To account for the low-bandwidth global memory, another memory is available for utilization. *Shared memory* is a very fast, on-chip memory, with a bandwidth up to 100× higher than global memory's one. It is only accessible from all threads of each block and lasts for the duration of a block execution. Its storage capacity is fairly low, therefore requires special treatment by the programmer. A usual technique for large matrices, that will be further explained and shown in practice, is to divide them in small *tiles*, meaning smaller individual matrices. Each tile can then be copied to shared memory and used as many times as is necessary. In cases where the kernel is memory-bound its usage can offer a huge speedup.

2.3.5 Local Memory

Local memory is a thread-private memory, stored, off-chip, in device DRAM [47]. The term *local* refers to a variable being private for every thread and not to the physical location. Because of its location, local memory is a slowly accessed memory, thus it can introduce a crucial performance bottleneck, depending on the amount of its usage during execution and whether it is cached or not.

When an array is accessed via a variable or dynamic indexes in a dynamic rather than static way, the whole array is placed on local memory, by the compiler. For example a

one-dimensional array `a` of size 256 is accessed using static indices, when it is defined as `a(1)=2; a(2)=5`, whereas in the following initialization: `a=2` or `a(j)=5`, is accessed in a dynamic fashion and thus matrix `a` is forced to be allocated in local memory. Moreover, when register memory is exceeded by scalar variables used inside a kernel, these are allocated in local memory. The phenomenon where some per thread variables cannot be placed in registers is called *spilling* and in some cases of devices with compute capability 1.x introduces a serious performance bottleneck. For devices of compute capability 2.x and higher local memory spilling does not introduce any performance degradation, since local memory is cached in the L1 cache.

2.3.6 Register Memory

Register memory is a thread-private, on-chip memory that is partitioned among all resident threads on a multiprocessor [47]. All variables declared locally inside a kernel, but not in shared memory, with the attribute `shared`, are placed either in register or local memory. The decision where a scalar thread-private variable will reside is made by whether there is adequate space in register memory, or elsewhere it will be placed on local memory. For a thread-private array it is ambiguous whether or not it will be placed in registers and depends on the size of the array and the way it is accessed.

Register memory is about 10 times faster than shared and as aforementioned local memory is another alternative to access global memory, which is a slowly accessed memory. As aforementioned, in cases where a thread-private variable is placed on local memory instead of register, the whole kernel execution time degrades.

In Table 2.4 the different types of memories that a device has are presented, among some important information a programmer should have in mind when composing a kernel. In table 2.5 some memory characteristics for the GPUs used for this diploma thesis are presented.

Memory	Location	Device Access	Scope	Lifetime
Register	On-chip	Read/write	One thread	Thread
Local	DRAM	Read/write	One thread	Thread
Shared	On-chip	Read/write	All threads in block	Thread block
Global	DRAM	Read/write	All threads and host	Application
Constant	DRAM	Read	All threads and host	Application
Texture	DRAM	Read	All threads and host	Application

Table 2.4: Device memories characteristics.

Name	Global memory	Constant memory	Shared memory
NVidia Quadro M2200	4 GB	64 kb	48 kb
NVidia Titan Xp	12 GB	64 kb	48 kb

Table 2.5: Memory features of the GPUs that were used.

2.4 CUDA API

CUDA, as aforementioned, is a parallel computing platform and application programming interface (API) model created by Nvidia. In computer programming, an API is a set of subroutine definitions, communication protocols, and tools for building software [51]. Generally, it is a set of clearly defined methods of communication among various components. A well structured API makes it easier to develop a computer program by providing all the building blocks, which are then put together by the programmer.

2.4.1 Syntax

A kernel is launched by the name of the kernel, two integers in the triple triangle braces –called chevrons– and the passed parameters inside parentheses. The first integer inside the triple chevron specifies the number of blocks to launch, whereas the second the number of threads per block. These two integers are separated by a comma. Afterwards, inside parenthesis the parameters and arrays, that the kernel needs to have access to, are listed and separated by comma, likewise to a common subroutine. The whole command has as follows: `kernelName<<<NumBlocks,NumThreads>>>(Param1,Param2,array1)`.

For a kernel to be executed, according to CUDA Fortran API, necessarily must be contained inside a *module*. Thus, at the beginning of the program, before any parameters are declared, a call to the module containing the kernel or kernels is needed. Is is also essential to include the command `use cudafor`, which connects the libraries for CUDA Fortran.

The passed arrays or parameters, inside of the parenthesis of the launched kernel, must be allocated to global memory. No host array can be passed (unless unified memory is utilized). To allocate an array or parameter on global memory, the declaration `device` must be used, as for example in `real(8), device, allocatable, dimension(:) :: u_d`. It is common to name the device parameter with the host name, followed by `_d` to distinguish it.

All data than should be used by the kernel must be copied from host to device before kernel is launched, and afterwards –if needed from the main program– they must be transferred back from device to host. This transfer in Fortran can happen using simply the equal sign, as in `u=u_d` to copy data from device to host. Another alternative is to use the function `cudaMemcpy()`, with the arguments to be the destination array, source array, and number of elements to be transferred (unlike the third argument of the CUDA C++ `cudaMemcpy()` which requires the number of *bytes* being copied). In CUDA Fortran there is no need to specify the direction of transfer, as the compiler is able to detect that, however, if the programmer desires, there is an optional fourth argument that specifies the direction of transfer, which takes on the values `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`.

The kernel, inside the module, is initiated by the argument `attributes(global) subroutine`, followed by the name and the parameters in parenthesis. To declare an array on global memory no specific attribute is needed. For example by writing `real(8) :: Y(:,:,:)` the array `Y` will reside on global memory. All parameters must have the `value` attribute, as in: `integer, value :: i`. To declare a parameter on shared memory the attribute `shared` is needed after the type declaration (e.x. `real(8),shared :: u_s(10)`). All parameters not declared with `shared` attribute will be transferred to registers.

2.4.2 Compiling

The CUDA compiler provides all necessary functions that are executed on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems integrated with multiple devices and many others. A script written in Fortran 90, that either contains CUDA API parts or not, can be compiled using the command `pgf90` for Linux or `pgf90.exe` for Windows. It is usually beneficial to compile the program using the architecture flag of the GPU in possession in order for the compiler to produce an executable suitable for the specific. This can be found by running the `pgaccelinfo` command in a terminal. The architecture flag can be found by the name `PGI Compiler Option:`, which in case of the Nvidia® Quadro M2200 GPU is the `-ta=tesla:cc50`. Some other important flags during compiling are shown in Table 2.6. All flags can be found using the command `pgfortran -Mcuda=help`.

Compiling the code with the flag `-Mcuda=ptxinfo`, important information about the different memories that are utilized by the program can be examined. The amount of bytes in constant memory can be seen by the keyword `cmem`. The usage of local memory can be observed with the keywords `lmem` for devices of compute capability 1.x, or `stack frame` for compute capability of 2.x and higher. Furthermore, to detect local memory spilling, running the same compiling flags, one can observe the bytes of spill loads and stores near the keywords `spill stores` and `spill loads`. It is important to note that these numbers refer to a statistic calculation and not to the generated code. Hence, the true amount of spilled loads and stores inside loops will be higher during code execution. In Figure 2.7 the output message of this command is presented for the code of the fifth chapter.

Flag	Description
`-Mcuda=fastmath`	Use fast math library (accuracy is limited)
`-Mcuda=ptxinfo`	Print informational messages from PTXAS
`-Mcuda=keepptx`	Keep PTX portable assembly files
`-Mcuda=maxregcount:<n>`	Set maximum number of registers to use on the GPU
`-Munroll=c:<n>`	Completely unroll loops with loop count n or less
`-Mcuda=lineinfo`	Generate GPU line information

Table 2.6: Important compiling flags.

```
ptxas info    : Compiling entry function 'poiseuilleflowkernels_velocitiesglobal_' for 'sm_50'
ptxas info    : Function properties for poiseuilleflowkernels_velocitiesglobal_
8 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 86 registers, 1024 bytes smem, 368 bytes cmem[0], 144 bytes cmem[2]
ptxas info    : Function properties for __internal_trig_reduction_slowpathd
80 bytes stack frame, 36 bytes spill stores, 36 bytes spill loads
```

Figure 2.7: Output message of -Mcuda=ptxinfo command.

2.4.3 Asynchronous Concurrent Execution

Apart from the extreme time saving that a kernel can offer, further parallelism can be achieved if consecutive operations correspond to the following cases:

- Computations on host
- Computations on device
- Data transfers from host to device memory
- Data transfers within the memory of a device
- Data transfers among different devices

These tasks can be executed further in parallel, offering another great advantage to total time shortening. For example if data transferring from host to device or vice versa are not needed for the kernel afterwards then these can be performed in parallel. Likewise if a set of data (ex. arrays) must be copied, then this can be performed asynchronous (by default data copies are performed synchronously). Moreover two independent kernels can

run concurrently, as well as a kernel and a subroutine on host. The degree of concurrency that can be succeeded in these cases greatly depends on the sources of the device, the data size and compute capability.

If different host kernels write on independent memory addresses then they can be launched asynchronously. In order to successfully launch asynchronous kernels on device, they must belong to different streams. A CUDA *stream* is a sequence of commands that are performed in order on the device [47, 2]. A stream can be created using the `cudaStreamCreate()` and destroyed with `cudaStreamDestroy()`. If the programmer wants to disable concurrent kernel launches, globally, he can set the environment variable `CUDA_LAUNCH_BLOCKING` to 1. It is wise to use that technique only for debugging purposes. Certain devices of compute capability 2.x and higher are able to run multiple kernels simultaneously. The level of parallel run –i.e. if the second asynchronous kernel will start exactly at the same time as the first one, or after a while–, depends on the amount of free resources on the device. A second kernel will be executed in parallel to the first one when resources become available or else it will run serially. After the completion of the previous kernels, one should take into consideration the hardware limits to predict, when the next kernel will start, such as the number of SMs and active warps for each kernel. The stream identity is included as a fourth parameter inside the triple chevrons. The whole syntax configuration is: `kernelName<<<blocks,threads,0,stream1>>>(x_d,y_d)`, where the 0 stands for the number of bytes allocated on shared memory.

As stated, asynchronous kernels require sufficiently ample resources. To give a more specific example in this case, running the command `pgaccelinfo`, recovers that the NVIDIA Quadro M2200 GPU has 8 multiprocessor with 2048 maximum threads per multiprocessor. Hence the maximum number of threads running in parallel on GPU device is $8 \times 2048 = 16384$. If one wants to run two kernels concurrently with the same amount of threads the maximum number of threads per grid is $16384/2 = 8192$. For best efficiency it is wise to choose number of threads per block 256 (multiples of 32, which is the number of threads of a single warp –see subsection 2.2.1– and also not to have too low occupancy –see subsection 2.6.4–), thus a reasonable number of blocks is $8192/256 = 32$ blocks of

256 threads per block. For 5 kernels fully running in parallel the threads per grid would be $16384/5 = 3276.8 \approx 3276$. If more than the aforementioned maximum threads per grid in each case are used then the kernels will not run fully in parallel, but will be executed when multiprocessors become available. Using advanced GPUs guarantees better parallel execution capabilities. For example the NVIDIA Titan Xp is equipped with 30 multiprocessors, each one of which can accommodate 2048 threads ($30 \times 2048 = 61440$ threads in total). For 2 completely parallel executed kernels the maximum number of threads per grid is $61440/2 = 30720$, whereas for 5 is $61440/5 = 12288$ threads.

Apart from parallel kernel execution, as aforementioned, memory transfers are also asynchronous is certain cases. First of all, data copies between a single device's memories are asynchronous. Data from global (or shared, constant, texture) memory are being transferred to each thread register, of a warp, simultaneously. Furthermore, memory copies from host to device are performed asynchronously, if a block of 64 KB or less is being copied, or if one of the commands `cudaMemcpyAsync()`, `cudaMemcpy2DAsync()` or `cudaMemcpy3DAsync()` is used for one-, two- or three-dimensional array, respectively. The asynchronous memory copies can be performed either on a set of data that needs to be copied, or between a copy and a kernel launch, if the GPU allows it. Whether a GPU supports parallel copy and kernel launch can either be determined by the `deviceOverlap` field of a `cudaDeviceProp` variable by running the `pgaccelinfo` command.

For the asynchronous transfer version it is required the data to be allocated on *pinned memory*; also named *page-locked memory*. The format of the memory asynchronous copies is: `istat = cudaMemcpyAsync(a_d, a_h, nElements, 0)`, where the additional argument introduced here is the stream ID. If the device is capable to perform memory copies and kernel launch asynchronously, and also if pinned memory is allocated, these two can be executed in parallel when they are assigned to different, nondefault streams (ie. non zero id stream).

In Figure 2.8 the path that data follow in order to be transferred to device from host are shown. When memory is allocated for variables that reside on the host, pageable memory is used by default [47]. In the first case of Figure 2.8 all data are stored in

pageable memory and have to first be copied to pinned memory and afterwards to device DRAM. However, if pinned memory is allocated directly, then data will be transferred immediately and faster.

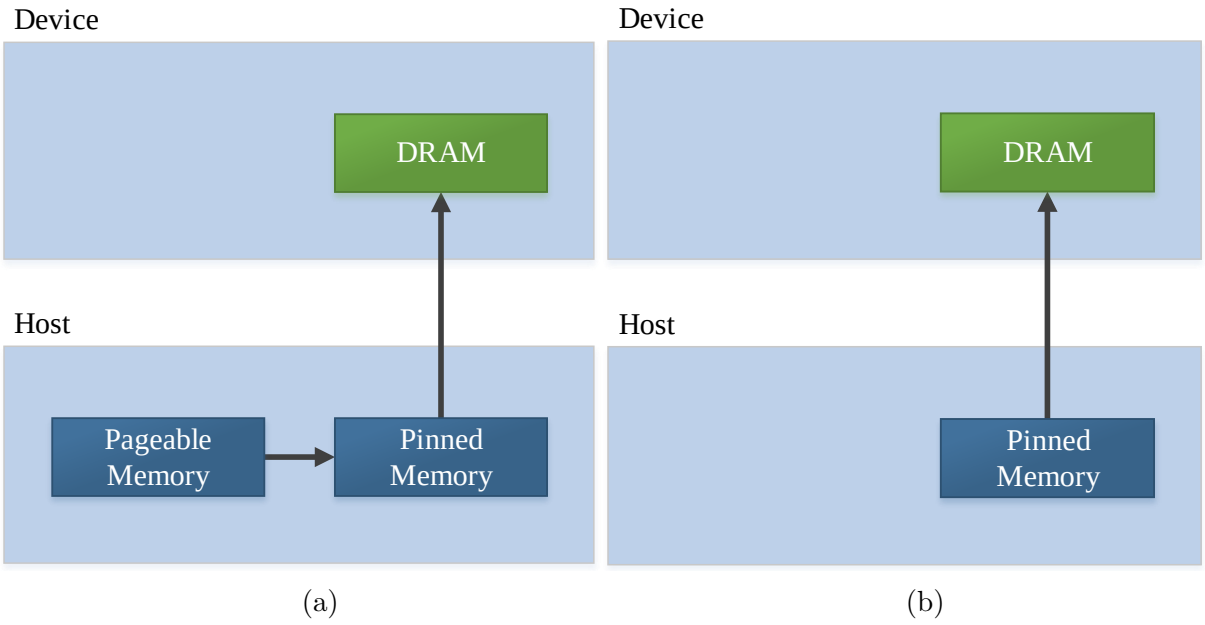


Figure 2.8: (a) Pageable data transfer and (b) pinned data transfer from host to device memory.

2.5 Parallelism Throttles and Performance Improvement

In computer architecture the Amdahl's law is a formula, that is used to compare a program's parallelization capabilities [52]. Specifically, it shows the theoretical maximum improvement or speed-up possible of a particular part of the serial code, by improving it using multiple processors. It is named after Gene Amdahl, a computer architect from IBM®. The speed-up index is independent of the API that is used to parallel execute a program, and as a result valid for CUDA implementation. The Amdahl's law is given by

the equation:

$$S_L = \frac{1}{(1-p)+\frac{p}{s}} \tag{2.2}$$

where S_L is the theoretical speed-up of the execution of the whole task, s is the speed-up of the part of the task that was parallelized and is always lower to 1 (for example if 5 second is the execution time of the parallelized part of the program that previously required 10 seconds, $s = 5/10$). Finally, p is the proportion of the execution time, that the part, benefiting from improved resources, originally occupied and is always greater to 1 (for example if the part of the parallel code takes 2 seconds, while in the original serial code it takes 5, the improvement would be 2/5).

This theoretical maximum is limited on GPUs by certain performance bottlenecks. In cases where these can be avoided performance is improved essentially. Thus, the programmer, in order to compose kernels of high performance, should examine carefully and always have in mind every performance throttle. Acknowledging the common problems of parallel execution, the programmer can guide his optimization techniques to speed-up the essential kernels. These problems, that one should try to eliminate, are the following:

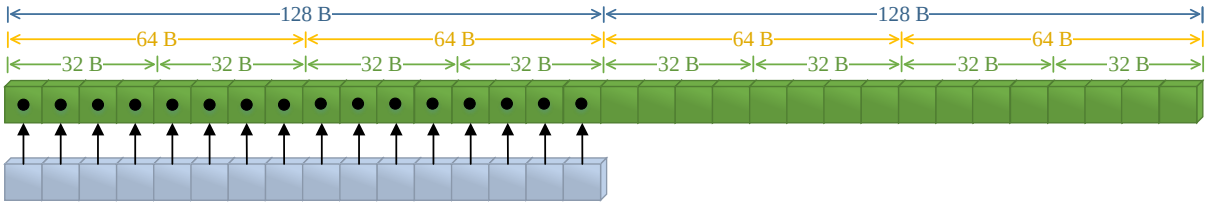
- Coalesced memory access in global memory
- Divergent branches
- Bank conflicts in shared memory
- Latency hiding

2.5.1 Memory Coalescing

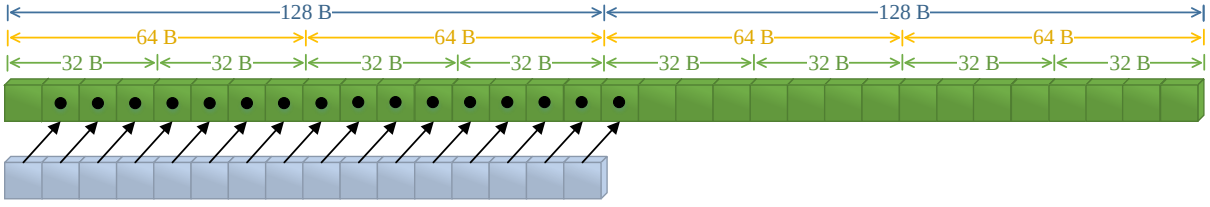
In CUDA Fortran all arrays are stored in memory per column as one-dimensional. That means that for a two-dimensional array, beginning with the first column all elements of every line, starting from 1 till the end, will be stored, and then similarly for the next columns.

When an array is allocated in global memory, either explicitly or implicitly, the array is aligned with a 256-byte segment of memory. Global memory can be accessed via 32-, 64-, or 128-byte transactions that are aligned to their size [47]. Optimal performance is met when threads of every warp access data in as few transactions as possible. Hence, when threads of a warp read or write, to or from an array that is stored on global memory, *coalesced access* must occur. This is schematically presented in Figure 2.9a – with a theoretical warp that consists of 16 threads–, where one transaction of 128 Bytes occurs. If even one thread does not access a sequential bank of the device memory, then *strided* or *uncoalesced* access takes place. This results in more than one transaction, as shown in Figure 2.9b. In this case the result will be three transactions per warp, one of 128 Bytes for the first half warp and two of 64 and 32 Bytes for the second half warp.

As aforementioned global memory is a very slow memory, thus for high performance kernels, transactions must be minimal. In cases where data must be read from different places of global memory from a single warp, then uncoalesced access can be avoided using shared memory, as it does not suffer from this particular bottleneck [12].



(a)



(b)

Figure 2.9: Schematic representation of a warp (here consisting of 16 threads) accessing global memory, with (a) coalesced access and (b) uncoalesced or strided access.

2.5.2 Warp Divergence

According to SIMT all threads of a warp must execute the same instruction at a time. Therefore, optimal efficiency is met when all threads follow the same execution path. Although, if a conditional branch forces certain threads of a warp to follow different path, i.e. to perform different instructions, then these different instructions will be performed one after the other, serially, resulting in what is called *warp divergence*. Warp divergence, also known as thread or branch divergence, can only occur within a warp. Different warps are executed independently, regardless of whether they are executing same or different code paths.

2.5.3 Bank Conflicts

Like every memory on device, data from shared memory must be transferred to registers before they can be used by the threads. Threads of a warp make requests to shared memory simultaneously. Every one of the 32 threads has the capability to request a different address, depending on the access pattern, and all threads can get their data simultaneously if the access pattern is optimal.

Shared memory transfers have the highest performance, when there is only one request for each bank per thread. In Figure 2.10a every thread requests data from a different bank. However, when two or more threads of a warp request different values from the same bank in a single request, then requests are serialized and performance degrades, resulting in *bank conflicts*, as shown in Figures 2.10c and 2.10d. In an alternative case where all threads request the same value, for example word one, of a bank then that value will be read only once and then will be *broadcasted* to all other threads that made the same request. A broadcast case is shown in Figure 2.11d. A subcase of broadcast, shown in Figure 2.11c is when not all, but several threads request the same value, then, as before, the value will be read only once and passed to the other threads, resulting in a *multicase*. Multicase is only available for devices with compute capability of 2.x and above. In conclusion, when more than one thread requests a single word from the bank will result in either a broadcast

or multicase, whereas if the request is done for different words, then the result will be a bank conflict. It is important to note that as warps are groups of threads of each block, accessing patterns that would result in bank conflict, but are of different blocks, will not have any affect whatsoever.

One way to be sure that no bank conflicts will occur is to have every thread access shared memory bank based on its identity `threadIdx%x`. If the same value from one bank is needed, then one should be careful to have all threads request the same word. Sometimes bank conflicts may not have any serious affect on the total execution time of a kernel, because if enough threads are running on an SM, the scheduler may switch to another warp till the bank copies are fulfilled resulting in latency hiding. Nevertheless even the added time for some bank conflicts is minimal compared to the time needed to access L2 cache or global memory, as the scheduler may be able to hide the latency.

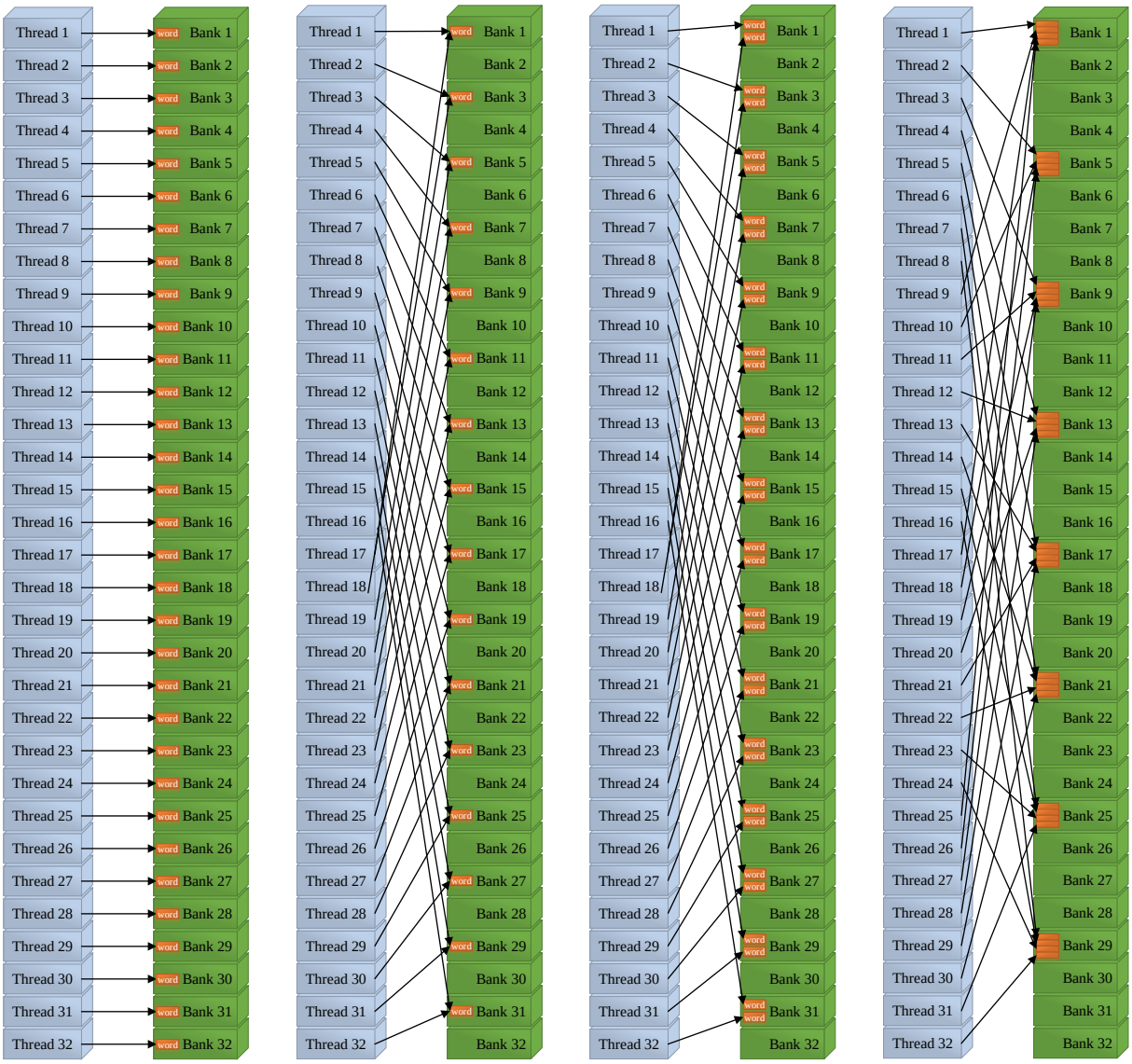


Figure 2.10: (a) Conflict free with stride of one bank accessing one 32-bit word per bank, (b) conflict free with stride of two banks accessing one 32-bit word per bank, (c) 2-way bank conflict due to stride of two banks accessing two 32-bit words per bank and (d) 4-way bank conflict due to stride of four banks accessing four 32-bit words per bank.

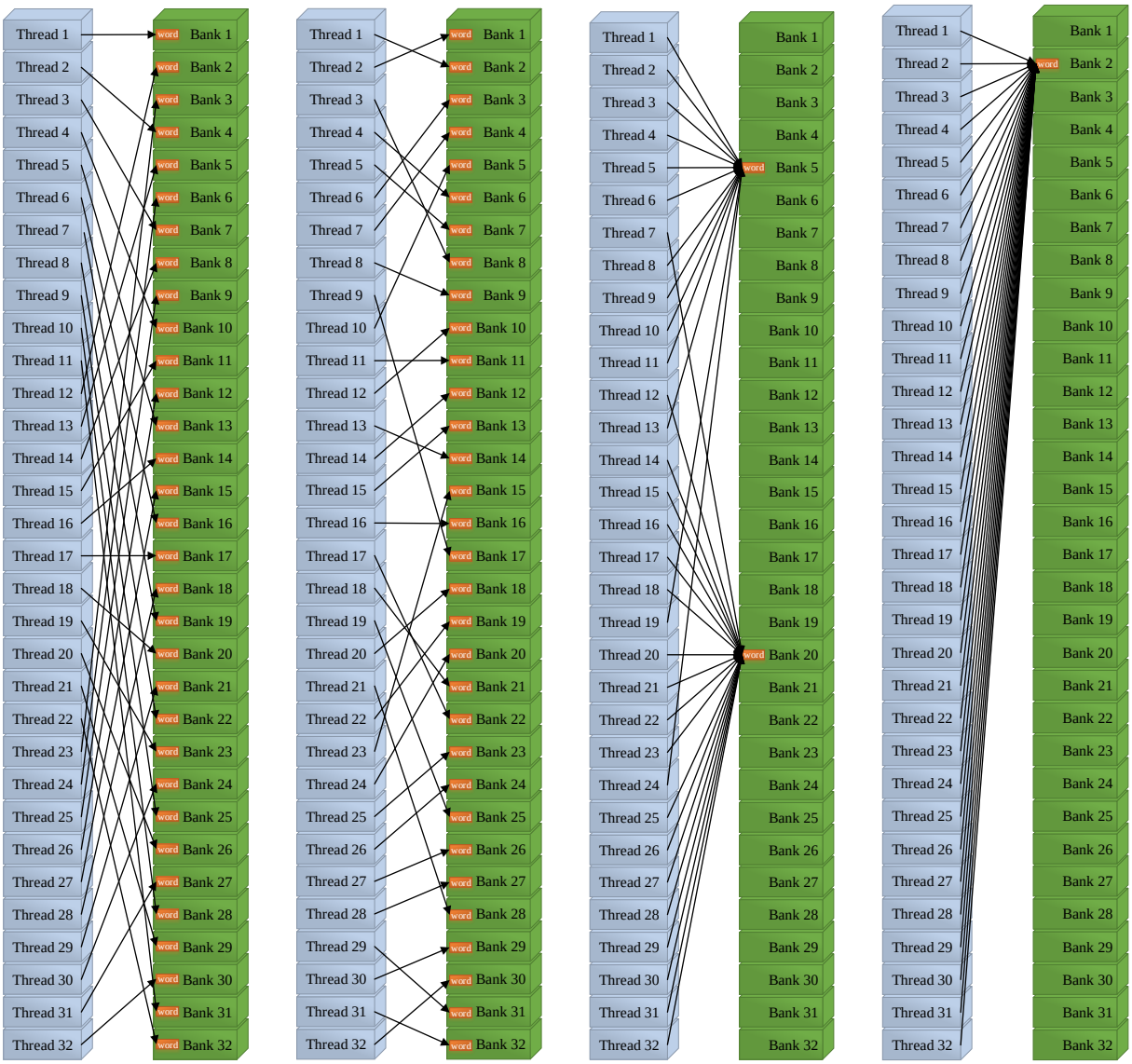


Figure 2.11: (a) Conflict free with stride of three banks accessing one 32-bit word per bank, (b) conflict free access of one 32-bit word per bank via random permutation, (c) conflict free multicase since threads access one 32-bit word per bank and (d) Conflict free broadcast since threads access one 32-bit word per bank.

2.5.4 Latency Hiding

The number of clock cycles that takes a warp to be ready to execute its next instruction, from the time the previous instruction is assigned, is called *latency* [2]. When the warp scheduler uses all resources, ie all available SPs, and has more instructions to issue to

warps at every clock cycle, then latency can be hidden. A GPU hides instruction latency by performing computations from other resident warps.

Two main types of instructions exist; arithmetic and memory instructions. Arithmetic instruction latency is the time required for an arithmetic operation to start, till the result is computed. Memory instruction latency is the corresponding time between the assignment of a load or store operation and data being successfully transferred. Regarding the necessary clock cycles, an arithmetic operation requires approximately 10-20 cycles, whereas an access to global memory requires 400-800 cycles.

The number of active warps in order to hide latency is provided by *Little's Law* [53], which is applied to queue theory, however can be successfully implemented to GPU parallel execution. This theorem states that the number of required warps is analogous to latency and throughput, according to:

$$Number\ of\ Requred\ Warps = Latency \times Throughput. \tag{2.3}$$

For example if only arithmetic and no memory operations are performed inside a kernel, which require approximately 10 cycles, then in order to keep a throughput of 40 warps per cycle, at least 400 warps are necessary to be completed per cycle. This is the reason why inside a kernel all unnecessary operations must be eliminated, in order to maintain a high computational throughput.

2.6 Metrics of Performance

Examining hardware limits is important when obtaining metrics of a CUDA program, to evaluate adequately its performance, relatively to the theoretical peak one. Although one should have in mind that substantial divergence from the hardware peak limits does not equal mediocre parallelism actions, but a kernel could be inevitably weighted enough with immense operations or memory transfers that positions an upper burden, which can

be much worse than the hardware's one.

A kernel can either be memory-bound or compute-bound. A memory-bound kernel spends most of its run time on memory instructions. A typical example of a memory-bound kernel is the addition of two arrays. In that case the only operation that takes place is an addition. Contrariwise, a kernel is compute-bound if most instructions that are performed are numerical operations. Placing a lot of long equations with many operations, will result in a compute-bound kernel.

Depending on whether a kernel is memory- or compute-bound, the programmer follows different path for optimization. The performance of a memory-bound kernel will greatly depend on optimization of memory accessing and data transferring, such as shared memory utilization, memory access coalescing etc., whereas on compute-bound kernel it is beneficial to reduce branch divergence and unnecessary operations.

The way that performance is measured and evaluated is according to the following metrics of performance:

- Total time
- Kernel execution time
- Speed-up
- Occupancy
- Bandwidth
- Throughput
- Efficiency

2.6.1 Total Time

The total time is the first thing that concerns when applying parallel programming and shows in practice the importance of CUDA API. Total time is measured for a device version of the code and it is usually compared with the equivalent host version, that runs on CPU. All operations are included, till solution is obtained, meaning that on device

version memory transfers from host to device and vice versa, as well as kernel launch and execution. This shows a direct comparison between a program running on CPU and GPU, presenting the essential benefits of parallel programming.

2.6.2 Kernel Execution Time

Another important time metric of performance is the execution time of each individual kernel. When experimenting different optimization techniques on a single kernel to improve its performance, the aim of every alteration is to maintain the run time as minimal as possible. In parallel programming what is of great importance is to reduce time with the same, correct output.

2.6.3 Speed-up

The overall speed-up of the parallel code compared to the serial one is a great indicator of the succeeded parallelism. It shows directly how faster the same task is accomplished using CUDA parallelization techniques. The speed-up of a single or more kernels is given by the following equation:

$$S = \frac{T_{CPU}}{T_{GPU}}. \tag{2.4}$$

2.6.4 Occupancy

2.6.4.1 Theoretical Occupancy

Occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM, as shown in Equation 2.5. Theoretical occupancy, introduces an upper limit of occupancy for each case, in order to compare the achieved

occupancy.

$$Occupancy = \frac{number\ of\ active\ warps\ per\ SM}{maximum\ possible\ number\ of\ active\ warps}. \tag{2.5}$$

Each device introduces certain limits during execution. For instance the NVIDIA Quadro M2200 GPU has a device limit of 32 blocks, 64 warps, 2048 threads being concurrently executed on an SM and also 65536 registers and 49152 bytes shared memory. These upper limits restrict theoretical occupancy. For example, a kernel launched with 63 registers per thread, 64 threads per block and 8 blocks in total will have a theoretical occupancy on this specific GPU of 50%, as the maximum number of blocks that can simultaneously execute on SM is limited by register usage. Specifically, $63 \times 64 = 4032$ registers per block, thus $65536/4032 \simeq 16$ blocks per SM. As the upper limit is 32 blocks, the theoretical occupancy would be $16/32 \times 100\% = 50\%$.

Increasing the occupancy does not necessarily guarantee improved performance. As seen from case study results there is, more or less, a golden line between the number of blocks and the number of threads [54]. Usually, optimal efficiency is met for 128, 256 or even 512 threads per block; however its greatly dependent on the application. Most of the times a "trial and error" technique will guide the programmer to peak performance.

2.6.4.2 Achieved Occupancy

Occupancy is a quantity that alters over time as warps begin and finish their execution, and can be different for each SM of the GPU. It can be affected by numerous factors, like the number of blocks launched. As the theoretical occupancy is calculated per SM, this result is not valid for a kernel launch with fewer blocks, than the minimum number of blocks that can run simultaneously on all SMs, at every clock cycle. The number of SMs of the device multiplied by the maximum active blocks per SM is called a *full wave* [55]. If the kernel is launched with less blocks than a full wave, the result will be an extremely low achieved occupancy.

In the previous example, the NVIDIA Quadro M2200 is equipped with 8 SMs, thus to achieve the theoretical occupancy of 50% it would require multiples of $8 \times 16 = 128$ blocks in order to have enough "waves" of warps to cover all SMs per clock cycle. Since the kernel is launched with only 8 blocks, the achieved occupancy is (8 active blocks)/(128 maximum theoretical blocks) $\times$ (0.5 theoretical occupancy)$= 0.03125 \simeq 3\%$. This explains the concept that in order for a GPU to give beneficial results, it should be fully utilized and loaded with many blocks and consequently many warps.

2.6.5 Memory Bandwidth

Bandwidth shows the rate at which data can be transferred. It is an extremely important factor that dictates the performance. Every change that is implemented to the code should be made according to how it affects the bandwidth. Bandwidth is greatly affected by the utilized memory in which data are stored, how the data are read and the order in which memory is accessed.

2.6.5.1 Theoretical Bandwidth

The theoretical bandwidth is the maximum bandwidth achievable by the hardware and can be calculated using the GPU's specifications available in the product literature. It can be calculated as follows:

$$BW_{theoretical} = 2 \left(\begin{array}{c} \text{Memory} \\ \text{clock rate} \end{array} \right) \left(\frac{\textit{Memory bus width}}{8} \right) \times 10^{-6}, \tag{2.6}$$

where the *memory clock rate* is measured in kHz and *memory bus width* is measured in Bits. The resulting bandwidth is given in $[GB/s]$.

Note that in the theoretical memory bandwidth calculation, the factor of 2.0 appears due to the double data rate of the RAM per memory clock cycle. Moreover, the division by eight converts the bus width from bits to bytes, and the factor of 10^{-6} handles the

kilohertz-to-hertz and byte-to-gigabyte conversions.

For the NVIDIA Quadro M2200 GPU the peak theoretical bandwidth is 88.1280 GB/s. No matter how many optimization techniques we apply the bandwidth of a kernel cannot overcome this upper limit.

2.6.5.2 Effective Bandwidth

Effective bandwidth is the measured bandwidth which a kernel actually achieves during execution, i.e. the amount of inputs and outputs in bytes over during the execution time of the kernel, and is calculated using the following equation:

$$BW_{effective} = \frac{(R_B + W_B)}{10^9\, t}, \tag{2.7}$$

where R_B is the number of bytes read per kernel, W_B is number of bytes written per kernel and t is the elapsed time given in seconds. The resulting bandwidth is in $[GB/s]$. For a memory-bound kernels the effective bandwidth is a metric to evaluate the performance. Higher bandwidth is translated to excellent performance.

2.6.6 Computational Throughput

Throughput is the amount of operations that can be processed per unit of time, commonly expressed as *GFlops*, which stands for billion floating-point operations per second [56]. Theoretical peak GFlops can be found in the GPU card handbook, although one should be careful as usually different values apply when operating on single or double precision. For example NVidia Quadro M2200 has a peak throughput of 2099 GFlops for single precision, however solely 65.60 GFlops for double precision; ratio 1:32 [57]. The NVidia Titan Xp card surpasses by far with a peak throughput of 12150 GFlops for single precision and 379.7 GFlops for double precision; ratio 1:32 [58].

In order to compute the actual throughput of the kernel one alternative is to stroll through the kernel and count the number of Flops per each thread. A general rule is to

count each operation or function (such as: +, -, *, **, /, sin, log, sqrt, exp, etc.) as 1 FLOP. Then the number of actual throughput can be calculated as:

$$T_{actual} = \frac{n_t \times FLOPS}{t} \times 10^{-9}, \tag{2.8}$$

where n_t is the number of threads per kernel, $FLOPs$ is the number of Flops per thread and t is the elapsed time in seconds.

Another way to compute a kernel's actual throughput is by utilizing a profiler (see section 2.6.8). Using a profiler can give the total number of Flops per execution, meaning that provides the user with the value of $n_t\times$ FLOPs. Thus measuring execution time one is able to compute the actual throughput much faster.

2.6.7 Branch efficiency

A branch in a computer program is an instruction, that can cause the execution of a different instruction sequence and as a result deviate from the default instructions order. Branch efficiency is an indicator of divergent branches inside a kernel. Specifically, its the ratio of divergent branches to total branches and can be calculated according to the following formula [12]:

$$Branch\ Efficiency = 100 \times \left(\frac{\#Branches - \#Divergent\ Branches}{\#Branches} \right). \tag{2.9}$$

A branch efficiency of 100% indicates that no divergent branch exists, whereas on the contrary zero efficiency indicates the existence of only divergent warps. It is important to note that sometimes the CUDA compiler may perform an optimization by itself, replacing the divergent branch instructions by what is called *predicated* instructions, for short segments of the code [12]. When this branch predication occurs, a variable is assigned for each thread, which takes only the values 0 or 1, according to the conditional, that intro-

duces the divergence. As a result, threads of a warp execute one path, that includes both conditionals, however the instructions are only performed by the threads with predicate of 1. In such case no branch divergence is reported. This optimization is expected to occur in cases the code meets certain criteria. One of the main ones is the divergence branch's size. Therefore, large divergent code is not expected to be optimized by the compiler. In Table 2.7 the peak hardware performance is shown for both GPUs used in this diploma thesis.

Name	**Memory bandwidth**	**Computational throughput**
NVidia Quadro M2200	88.1 GB/s	2099(sp*)/65.6(dp**) GFLOPs
NVidia Titan Xp	547.7 GB/s	12150(sp*)/379.7(dp**) GFLOPs

Table 2.7: Performance features of the GPUs that were used *(*single precision, **double precision).*

2.6.8 Acquiring Metrics and Bottleneck Resolving

Evaluating a kernel, using the the aforementioned metrics, is vital to examine and further understand its behavior during operation, from the simplest –determining if a kernel actually runs on a GPU– to the most complicated –eliminating bottlenecks to increase performance–. There are many options and tools available to collect all necessary metrics and many more.

Comparing between the total time of a code version running on host and on device can be done via CPU timers. The current time is stored in a variable by calling the function `cpu_time()`, declaring the beginning of each case. At the end, after an iteration loop for example, another record of the time is taken and stored in a variable calling the same function. The difference between these two variables is the total execution time of each method (host and device version), till solution.

When measuring a kernel's execution time, CPU timers cannot be utilized, due to the asynchronous nature of multiple threads, as the first thread to reach the second timer would stop it, before the others finish their operations and thus measuring an erroneous total time. Another alternative one may think, is using host-device synchronization

points, such as `cudaDeviceSynchronize()`, before a CPU timer, however this introduces the problem that they stall the GPU pipeline, thus this technique cannot be used to synchronize the threads.

CUDA offers alternatives to CPU timers for kernels. Timing of each kernel can be achieved with two approaches, both giving valid results. The first method, which is hard-coded and thus requires adding command lines inside the program, is done via the CUDA event API. The programmer is capable to create and destroy events, record events, and compute the elapsed time in milliseconds between two recorded events. The CUDA events make use of the concept of CUDA streams (see subsection 2.4.3). CUDA events are of type `cudaEvent_t` and can be created and destroyed with the commands `cudaEventCreate()` and `cudaEventDestroy()`, respectively. Additionally, using `cudaEventRecord()` places the start and stop events into the default, null stream. The device will record a time stamp, when it reaches that event in that specific stream. The function `cudaEventSynchronize()` blocks CPU execution until the specified event is recorded. At the end, the function `cudaEventElapsedTime()` returns in the first argument the number of milliseconds elapsed between the recording of start and stop. This value has a resolution of approximately one half microsecond, thus for every re-run of the kernel different time values will likely be presented. The second approach is via profilers.

Since CUDA Toolkit 5, a very powerful tool is available; the *GPU Profiler*. Running the program executable with the command `nvprof` that is a universal profiler of CUDA kernels for every language, as long as the program uses CUDA API. PGI also has its own version of profiler, the command-line profiler, `pgprof`, with a version of Visual profiler for easier kernel examination [59]. The profilers are useful tools to examine a kernel individually, such as every memory copy performances, as well as many other important characteristics of device kernels that can be found for example in [46] for PGI profiler or in [2] for NVidia profiler. In Appendix B is thoroughly presented the way to obtain the metric using the PGI profiler. Finally, other debugging options also exist such as the `cuda-memcheck`, which checks for memory violations.

Chapter 3

Heat conduction in an L-shaped fin

Initially, CUDA Fortran's capabilities are being examined by studying a problem of heat conduction. Through this primal chapter some CUDA parallelism techniques are being examined, as well as the different metrics that completely describe the succeeded level of performance. The results of parallelism on GPUs showed excellent improvement compared to CPU performance, even using primal parallelization techniques.

First of all, the description of the problem is presented, as well as the governing equations for a small total number of nodes and afterwards the conclusive set of equations is adjusted to a more general case of arbitrary dimensions and nodes. Later on, the approach to solve it is thoroughly examined. In the latter section the implementation of the solving method on host is demonstrated and afterwards the different cases on device, in order to eliminate parallelism bottlenecks (as described in section 2.5) and subsequently achieve faster convergence to the final results.

There are several methods in order to obtain the numerical formulation of a heat conduction problem. These are the finite difference method, the finite element method, the boundary element method and the energy balance (or control volume) method [60]. Each one of them introduces its own advantages and disadvantages. In this chapter the usage of the energy balance approach will be discussed, in order to derive the governing equations. This method is based on formulating the energy balances on control volumes instead of heavy mathematical formulations, that the aforementioned alternative methods

require. The final result is indeed the same set of algebraic equations as the finite difference method.

The examined case is a problem of heat conduction that occurs inside an L-shaped fin. By making the assumption that the depth of the fin is considerably larger than the rest dimensions, solving for a two-dimensional domain, produces correct results. Moreover, the numerical formulation and solution of the problem is set for steady state conditions. In the developed program, the user is able to specify arbitrary dimensions for the fin surfaces and also the total number of nodes in each direction.

3.1 Problem Description

For this problem the dimensions were chosen to be relatively small, analogous to a CPU fin. In Figure 3.1 a schematic representation of the fin is presented. The examined fin is L-shaped with 40 mm in width and 12 mm in height. All dimension parameters are kept in relevance to n_1, altering of which determines the total number of nodes of each case. Subsequently, dimension m_1 is equal to $n_1 - 1$, n_2 equal to $5(n_1 - 1) + 1$ and m_2 equal to $m_1 + 1$. As the height and width of the fin are kept constant, the parameters dx and dy are determined as $0.08/(n_1 - 1)$ and $0.012/(m_1 + m_2 - 1)$ respectively, in order to keep the dimensions at the same initial values.

The material of the fin was chosen to be *Aluminum 6061*, an alloy widely used for heat sink fins with a thermal conductivity of approximately $k = 166\ W/mK$ [61]. All surfaces of the fin are subjected to heat convection at an atmospheric temperature of $T_\infty = 288K$, except the left side, which is subjected to a heat flux at a uniform rate $\dot{q}_R = 40000\ W/m^2$. Stating that air travels vertically to our two-dimensional fin, and assuming a very long plate in the third direction compared with the other two, a reasonable estimation for the heat conduction coefficient for turbulent flow is $h = 80\ W/m^2K$.

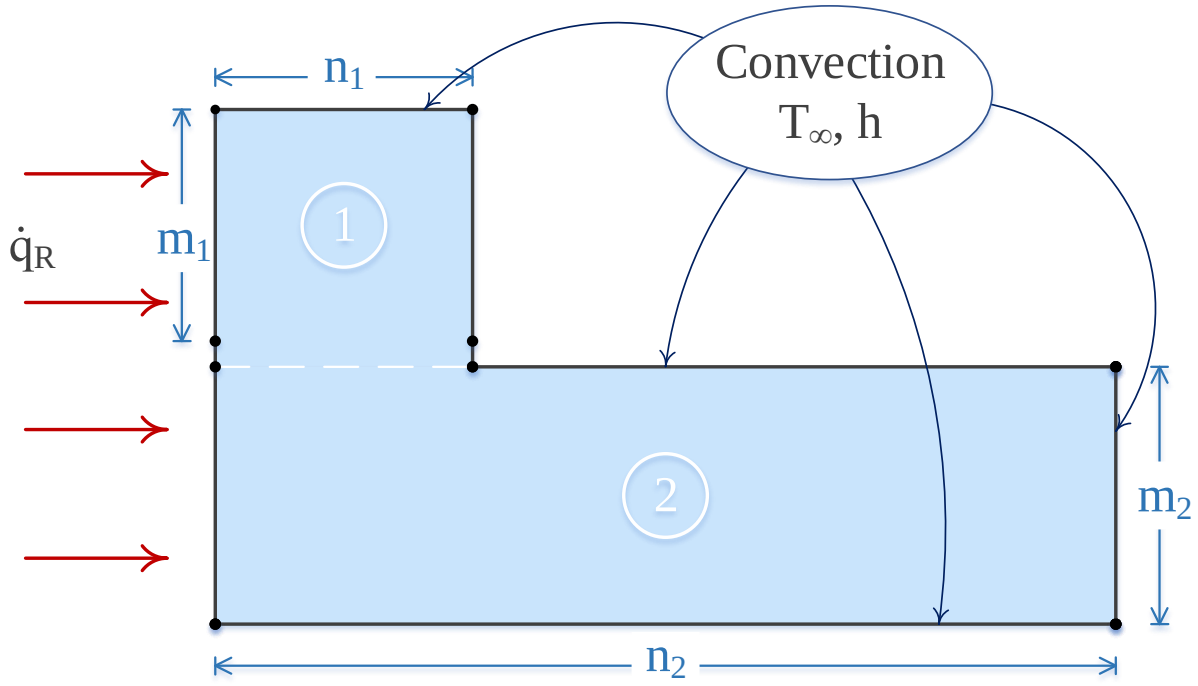


Figure 3.1: Schematic representation of the studied L-shaped fin.

3.2 Formulation

As aforementioned, to derive the solving equations for the problem, the energy balance approach is applied. The energy balance for a control volume can be generally expressed as follows:

$$\begin{pmatrix}\text{Rate of}\\ \text{change of}\\ \text{heat in c.v.}\end{pmatrix} = \begin{pmatrix}\text{Rate of}\\ \text{heat flow}\\ \text{in to c.v.}\end{pmatrix} - \begin{pmatrix}\text{Rate of}\\ \text{heat flow}\\ \text{out of c.v.}\end{pmatrix} + \begin{pmatrix}\text{Rate of heat}\\ \text{generation}\\ \text{in c.v.}\end{pmatrix} - \begin{pmatrix}\text{Rate of heat}\\ \text{consumption}\\ \text{in c.v.}\end{pmatrix}. \quad (3.1)$$

Assuming that the direction of heat conduction is towards the volume element of the node that is examined, at all around surfaces, the term of heat flow out of the control volume is omitted. Moreover, no heat is generated or consumed, thus the relevant terms are also zero, and since steady state conditions are studied the energy balance on the

volume element is simplified to:

$$\dot{Q}_{cond,left} + \dot{Q}_{cond,right} + \dot{Q}_{cond,top} + \dot{Q}_{cond,bottom} = \frac{dE}{dt} = 0. \tag{3.2}$$

In equation 3.2 the term $\dot{Q}$ represents heat flux and E the total energy of the volume element. Heat conduction, according to the differential form of Fourier's law, can be expressed as:

$$\dot{Q}_{cond} = -k\frac{dT}{dx}, \tag{3.3}$$

where k is the thermal conductivity of the material of the control volume, T is the temperature and x is the spatial coordinate in the direction of heat flow.

In numerical studies the direct determination of the temperature gradient is incredibly time consuming and thus it is usually approximated as a linear difference between two adjacent nodes. If the nodes are located close enough to each other, the introduced error of this assumption is tolerable and negligible.

To obtain the gradient of the temperature at the right surface of the volume element, the Taylor's series for upstream differences is written:

$$T(x)|_{i,j+1} = T(x)|_{i,j} + \left.\frac{dT(x)}{dx}\right|_{i,j} \Delta x + \left.\frac{d^2T(x)}{dx^2}\right|_{i,j} \frac{\Delta x}{2} + O(\Delta x^3). \tag{3.4}$$

Solving for the first derivative and keeping the accuracy to $O(\Delta x^2)$ equation 3.4 is formulated as:

$$\left.\frac{dT(x)}{dx}\right|_{i,j} = \frac{T(x)|_{i,j+1} - T(x)|_{i,j}}{\Delta x} + O(\Delta x^2) \tag{3.5}$$

Thus heat conduction can be sufficiently approximated by the following equation:

$$\dot{Q}_{cond} = -k\frac{T(x)|_{i,j+1} - T(x)|_{i,j}}{\Delta x} + O(\Delta x^2). \tag{3.6}$$

Similarly, the temperature gradients towards the other directions can be derived. As a result for each element volume of the L-shaped fin, the energy balance can now be written.

One might be thinking that if heat is conducted into the element from both sides, as assumed in the formulation, the temperature of the medium will have to rise and thus heat conduction cannot be steady. Perhaps a more realistic approach would be to assume the heat conduction to be into the element on the left side and out of the element on the right side. If one repeats the formulation using this assumption, the same result will be obtained, since the heat conduction term on the right side in this case involves $T(x)|_{i,j} - T(x)|_{i,j+1}$ instead of $T(x)|_{i,j+1} - T(x)|_{i,j}$, which is subtracted instead of being added. Therefore, the assumed direction of heat conduction at the surfaces of the volume elements has no effect on the formulation. Besides, the actual direction of heat transfer is usually not known. However, it is convenient to assume heat conduction to be into the element at all surfaces and neglect the sign of the conduction terms. Then all temperature differences in conduction relations are expressed as the temperature of the neighboring node minus the temperature of the node under consideration, and all conduction terms are added.

3.3 Governing equations

Firstly, the equations for a fin of specific dimensions are derived with few nodes. Later on the formulation of the equations for a generic L-shaped fin of arbitrary dimensions is accomplished. The fin is subjected to convection, as it is exposed to air at ambient conditions, and a constant heat flux at its left side surface. The problem is schematically presented in Figure 3.2. Inner nodes are subjected only to conduction, due to their

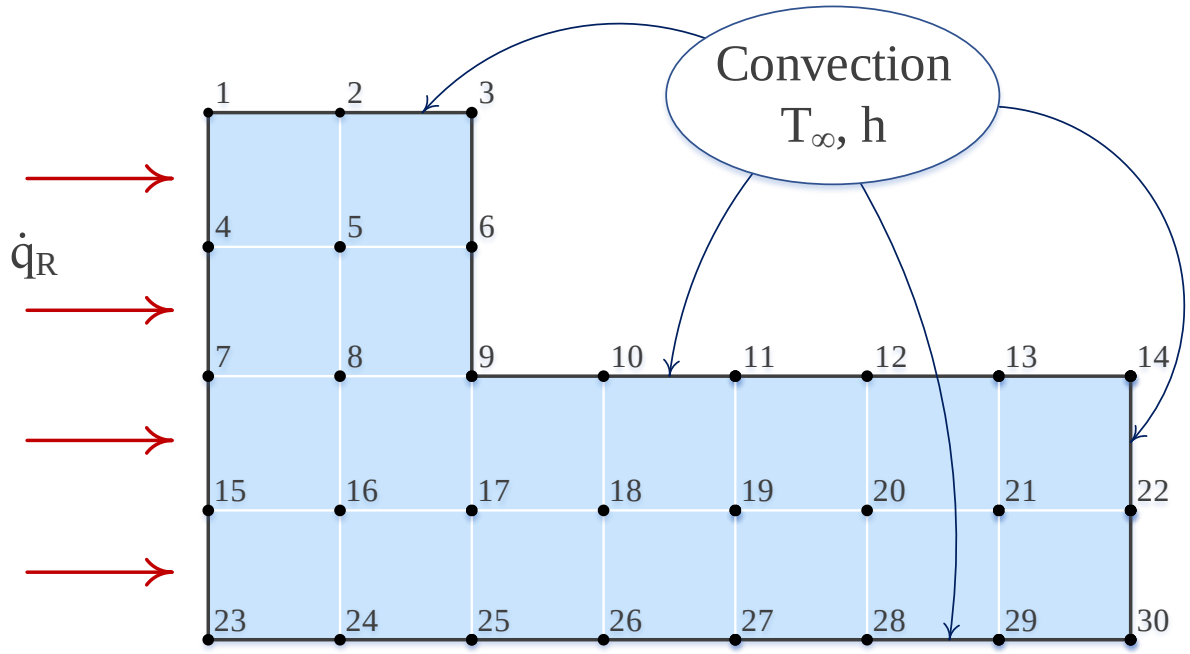


Figure 3.2: Schematic representation of an example L-shaped fin.

surrounding volumes. In Figures 3.3 and 3.4 the heat fluxes on the volume elements of certain essential nodes are shown.

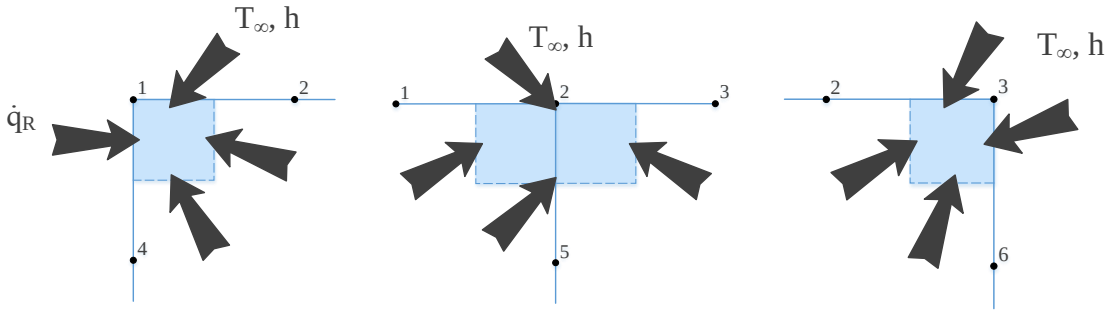


Figure 3.3: Schematics for energy balances on the volume elements of nodes 1, 2 and 3.

Node 1: The volume element of the first corner node is subjected to convection at the top surface, heat flux at the left surface and conduction at the bottom and right surfaces.

From an energy balance of this element it is obtained:

$$h\frac{\Delta x}{2}(T_\infty - T_1) + \dot{q}_R\frac{\Delta y}{2} + k\frac{\Delta x}{2}\frac{T_4 - T_1}{\Delta y} + k\frac{\Delta y}{2}\frac{T_2 - T_1}{\Delta x} = 0,$$

$$\left(\frac{h\Delta x}{2} + \frac{k\Delta x}{2\Delta y} + \frac{k\Delta y}{2\Delta x}\right)T_1 - \frac{k\Delta y}{2\Delta x}T_2 - \frac{k\Delta x}{2\Delta y}T_4 = \frac{\Delta y}{2}\dot{q}_R + \frac{h\Delta x}{2}T_\infty. \tag{3.7}$$

Node 2: The volume element of this boundary node is subjected to convection at the top and conduction at the bottom, left and right surfaces. An energy balance on this element gives:

$$h\Delta x(T_\infty - T_2) + k\frac{\Delta y}{2}\frac{T_3 - T_2}{\Delta x} + k\Delta x\frac{T_5 - T_2}{\Delta y} + k\frac{\Delta y}{2}\frac{T_1 - T_2}{\Delta x} = 0,$$

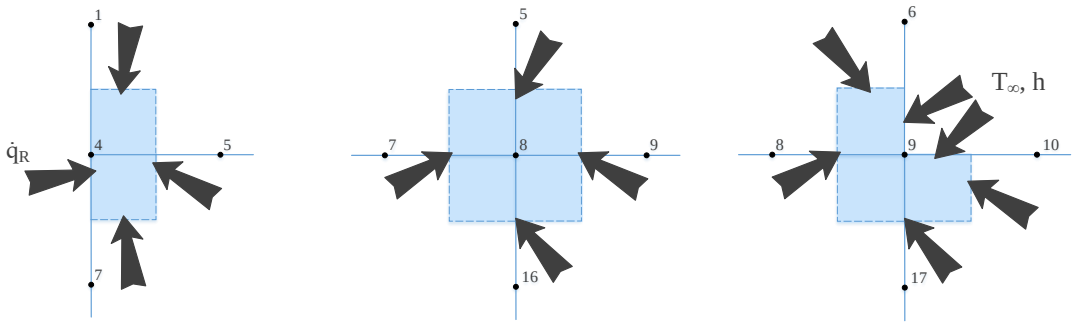
$$\left(h\Delta x + \frac{k\Delta y}{\Delta x} + \frac{k\Delta x}{\Delta y}\right)T_2 - \frac{k\Delta y}{2\Delta x}T_3 - \frac{k\Delta y}{2\Delta x}T_1 - \frac{k\Delta x}{\Delta y}T_5 = h\Delta x T_\infty. \tag{3.8}$$

Node 3: The volume element of this corner is subjected to convection at the top and right surfaces and to conduction at the left and bottom surfaces. Writing the energy equation:

$$h\left(\frac{\Delta x}{2} + \frac{\Delta y}{2}\right)(T_\infty - T_3) + k\frac{\Delta x}{2}\frac{T_6 - T_3}{\Delta y} + k\frac{\Delta y}{2}\frac{T_2 - T_3}{\Delta x} = 0,$$

$$\left(\frac{h\Delta x}{2} + \frac{h\Delta y}{2} + \frac{k\Delta x}{2\Delta y} + \frac{k\Delta y}{2\Delta x}\right)T_3 - \frac{k\Delta y}{2\Delta x}T_2 - \frac{k\Delta x}{2\Delta y}T_6$$

$$= h\left(\frac{\Delta x}{2} + \frac{\Delta y}{2}\right)T_\infty. \tag{3.9}$$



(a) (b) (c)

Figure 3.4: Schematics for energy balances on the volume elements of nodes 4, 8 and 9.

Node 4: This node is subjected to heat flux at the left surface and conduction at the top, right and bottom surfaces. Identical conditions occur also at the rest nodes of the left boundary of the fin. The energy equation gives:

$$k\Delta y\frac{T_5-T_4}{\Delta x}+k\frac{\Delta x}{2}\frac{T_1-T_4}{\Delta y}+k\frac{\Delta x}{2}\frac{T_7-T_4}{\Delta y}+\dot{q}_R\Delta y=0,$$

$$\left(\frac{k\Delta y}{\Delta x}+\frac{k\Delta x}{\Delta y}\right)T_4-\frac{k\Delta y}{2\Delta x}T_5-\frac{k\Delta x}{2\Delta y}T_1-\frac{k\Delta x}{2\Delta y}T_7=\dot{q}_R\Delta y. \tag{3.10}$$

Node 5: This specific node is an interior one. Thus its volume element is subjected to conduction at all surrounding surfaces. Hence, the energy balance adds:

$$k\Delta y\frac{T_6-T_5}{\Delta x}+k\Delta y\frac{T_4-T_5}{\Delta x}+k\Delta x\frac{T_2-T_5}{\Delta y}+k\Delta x\frac{T_8-T_5}{\Delta y}=0,$$

$$\frac{\Delta y}{\Delta x}T_6+\frac{\Delta y}{\Delta x}T_4+\frac{\Delta x}{\Delta y}T_2+\frac{\Delta x}{\Delta y}T_8-2\left(\frac{\Delta y}{\Delta x}+\frac{\Delta x}{\Delta y}\right)T_5=0. \tag{3.11}$$

Node 6: The volume element of this node is subjected to convection at the right surface and conduction at the top, left and bottom surfaces. An energy balance on this element

gives:

$$h\Delta y(T_\infty - T_6) + k\frac{\Delta x}{2}\frac{T_3 - T_6}{\Delta y} + k\frac{\Delta x}{2}\frac{T_9 - T_6}{\Delta y} + k\Delta y\frac{T_5 - T_6}{\Delta x} = 0,$$

$$\left(h\Delta y + \frac{k\Delta x}{\Delta y} + \frac{k\Delta y}{\Delta x}\right)T_6 - \frac{k\Delta x}{2\Delta y}T_3 - \frac{k\Delta x}{2\Delta y}T_9 - \frac{k\Delta y}{\Delta x}T_5 = h\Delta y T_\infty. \tag{3.12}$$

Node 9: Special treatment is needed for the inner corner node, concerning the volume's surrounding surfaces. The volume element of this node is subjected to convection at the L-shaped exposed surface and to conduction at the other surfaces.

$$h\left(\frac{\Delta x}{2} + \frac{\Delta y}{2}\right)(T_\infty - T_9) + k\frac{\Delta y}{2}\frac{T_{10} - T_9}{\Delta x} + k\frac{\Delta x}{2}\frac{T_{17} - T_9}{\Delta y}$$

$$+k\Delta y\frac{T_8 - T_9}{\Delta x} + k\frac{\Delta x}{2}\frac{T_6 - T_9}{\Delta y} = 0,$$

$$\left(\frac{h\Delta x}{2} + \frac{h\Delta y}{2} + \frac{3k\Delta y}{2\Delta x} + \frac{3k\Delta x}{2\Delta y}\right)T_9 - \frac{k\Delta y}{2\Delta x}T_{10} - \frac{k\Delta x}{\Delta y}T_{17}$$

$$-\frac{k\Delta y}{\Delta x}T_8 - \frac{k\Delta x}{2\Delta y}T_6 = h\left(\frac{\Delta x}{2} + \frac{\Delta y}{2}\right)T_\infty. \tag{3.13}$$

The equations for rest of the nodes can be derived accordingly. Nodes 14 and 22 are similar to nodes 3 and 6 respectively. Nodes 23 and 30 are also similar to 1 and 3 with slight modifications. Moreover, equations for the nodes 10 till 13 and 24 till 29 can be easily derived from the equation of node 2.

The derived equations for the above specific case can be reconstructed to fit for a general case of arbitrary dimensions. The total nodes according to the schematic repre-

sentation of the fin of arbitrary dimensions are:

$$nodes = n_1 m_1 + n_2 m_2. \tag{3.14}$$

The aforementioned equations, that describe the problem, constitute a system of linear equations of the form:

$$AT = b. \tag{3.15}$$

T is a $(nodes \times 1)$ matrix containing the unknown temperature parameters, A is a $(nodes \times nodes)$ matrix that includes the constant coefficients of each temperature at each one of the aforementioned equations and b is a $(nodes \times 1)$ matrix with the right hand sides of these equations.

In the main code of this chapter the host subroutine that initializes the matrices A and b is included, by the name `initialize(A,b)` and is located inside the module `HeatConductionRutines`. Each element of the two matrices obtains values according to the set of equations. This subroutine is called at the beginning of each version before the iterations for convergence take place.

3.4 Solving Methods

In order to solve the linear system $AT = b$ two different approaches exist; the *iterative* and the *direct* method [61]. A method that requires a finite number of operations for computing the exact solution of the unknown variables of T is called direct method. Direct methods in most cases are prohibitively time expensive or even impossible to be applied when the number of total nodes is too big, regardless the computational power one might have available.

On the other hand, iterative are called the methods that utilize an initial guess and

through a sequence of approximate solutions converge to the one that is almost identical to the previous one. The convergence of the results is succeeded inside the limits of a tolerable error. This minimum tolerance is usually fixed to a magnitude of approximately 10^{-9} for double precision numbers and 10^{-6} for single precision. Iterative solutions are mostly preferred instead of the direct ones for big matrices, due to their intensely reduced time till convergence and in nonlinear cases are typically the only choice.

Many different iterative methods with a wide range of applications have been developed. Broadly known methods are the Gauss-Seidel, Jacobi, and others. The Gauss-Seidel method uses the already calculated solution, from the previous iteration, of the other unknown parameters, in order to calculate the following one, whereas Jacobi is strictly limited to the solution of the previous iteration. Although the Gauss-Seidel method converges faster than the Jacobi, it is not suitable for parallelism. CUDA is a thread oriented API and thus each thread does not have access to the results of the other threads. Ergo, the best choice that can be paralleled is the Jacobi iterative method.

Given the system $AT = b$ where:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \quad T = \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_n \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}. \tag{3.16}$$

Another way to write the matrix A of Equation 3.16 is to decompose it to the diagonal matrix ($diagA$) and the residual matrix (A').

$A = diagA + A'$ where

$$diagA = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & a_{nn} \end{bmatrix}, \quad A' = \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ a_{21} & 0 & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & 0 \end{bmatrix}. \tag{3.17}$$

As a result the solution can be obtained iteratively via:

$$T^{(k+1)} = diagA^{-1}\left(b - A'T^{(k)}\right). \tag{3.18}$$

In the above equation 3.18, $T^{(k)}$ is the approximate solution of the k^{th} iteration and $T^{(k+1)}$ is the solution of the next iteration. The above equation can also be written, according to elements and their indexes, as follows:

$$T_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{i \neq j} a_{ij} T_j^{(k)}\right). \tag{3.19}$$

Afterwards, three different versions will be presented. At the beginning, the implementation of the solving algorithm on CPU is discussed and subsequently three different cases on GPU.

3.4.1 Host version

The algorithm for the implementation of Jacobi iterative method on CPU was structured based on Equation 3.19 and is presented in Figure 3.5 as a flowchart. In that

direction, the general structure consists of two main *do*-loops, in order to implement the Jacobi iterative method and another one, that encloses these two loops, which checks if convergence to a solution has been reached. As aforementioned to solve with Jacobi method, an initial guess is necessary. As a result initialization of the T array is made at the beginning. Of course $Tnew$ array does not need a specific initial value since it does not participate in the solving equation, but to avoid any run-time errors it is initialized with a value.

One of the two enclosed loops is responsible to iterate over every line (i) of the matrices A and b and the other over every element of every column (j). The latter *do*-loop is located inside the first one. With these nested loops the calculation of the total *sum* of Equation 3.19 is accomplished. After the inside loop (j) terminates and before line (i) changes, the new value of temperature is calculated, by reducing the b_i element by the total *sum* and then deriving from the result the diagonal element of matrix A, a_{ii}. This sequence is repeated until one of the two conditions is satisfied, meaning that the iterations terminate if the maximum residual between the current and previous solution becomes smaller than the acceptable tolerance, in which case a solution has been succeeded, or the number of iterations exceeds a maximum limit, in which case either more iterations are needed or the method cannot converge, probably because the initial guess is too far from the actual solution.

Below a snippet of the host code is cited, with the outer main loop for convergence check and both the enclosed loops that perform the Jacobi method.

```
1:    do while (maxResidual >tol .and.iter <=iterMax)
2:        maxResidual = 0.0_fpKind
3:        do i = 1, nodes
4:            summation = 0.0_fpKind
5:            do j = 1, nodes
6:                if (j .ne. i) then
7:                    summation = summation + A(i,j) * T(j)
8:                end if
```

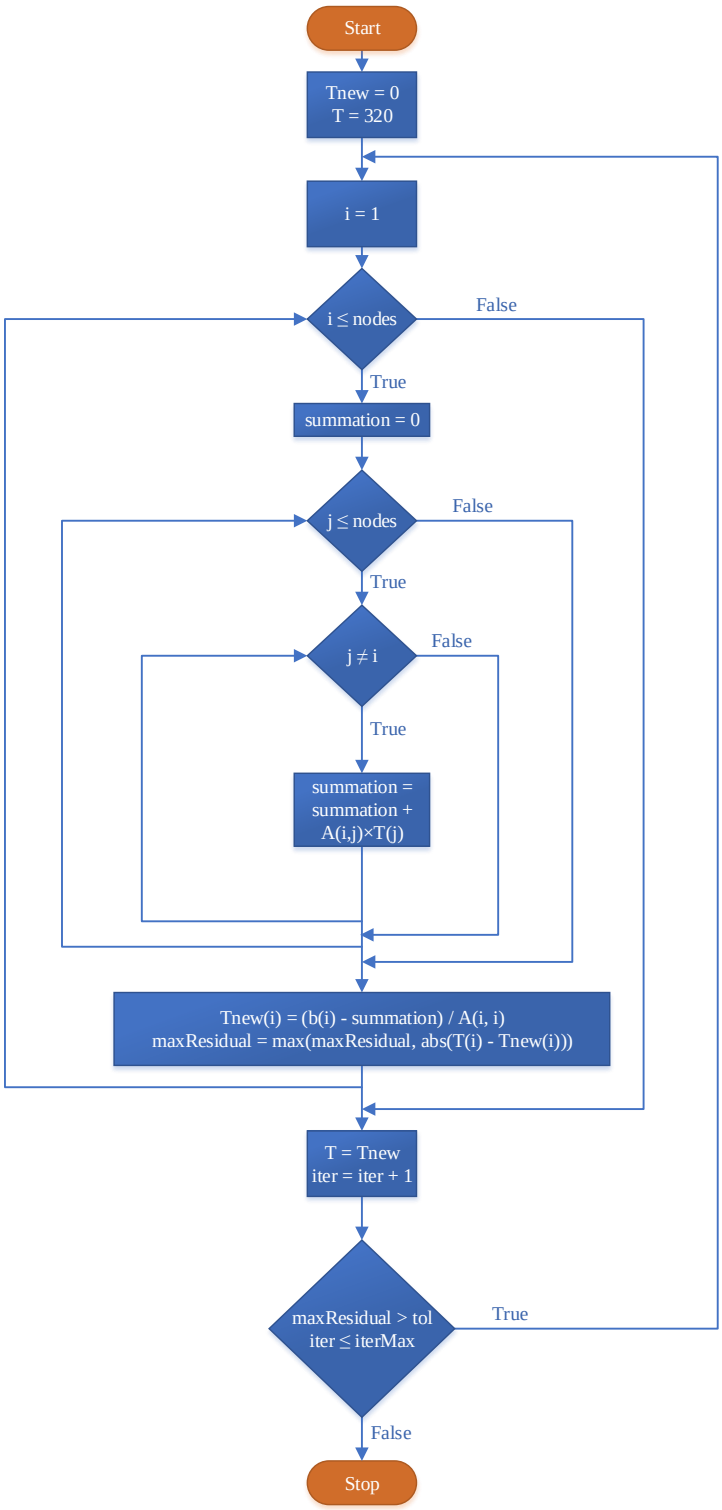


Figure 3.5: Flowchart of the host version

```
9:            end do
10:           Tnew(i) = (b(i) - summation) / A(i, i)
11:           maxResidual = max(maxResidual, abs(T(i) -Tnew(i)))
12:       end do
13:       T = Tnew
14:       if (mod(iter, reportInterval) ==0) write(1,*) iter, maxResidual
15:       iter = iter + 1
16:   end do
```

Listing 3.4.1: Host code of Jacobi iterative method.

3.4.2 Device version on global memory

Next step is to utilize CUDA API to accelerate the convergence. Implementing parallelism on the host version, the elimination of the outside *do-loop*, which runs for every line of the matrix A, can be effectively succeeded. A slight modification on this version has been made and that is the initial value of `summation` variable, which is set to `b(i)` instead of zero, and the solving equation is set to `Tnew(i)=Tnew(i)-A(i,j)*T(j)`. After the *do-loop*, the new temperature value is found by the equation `Tnew(i) = Tnew(i) / A(i,i)`. That has been made in order to reduce the number of mathematical operations that are executed, because too many operations slow down the whole kernel, as the computation load on each thread increases.

At the beginning, before the kernel is called, all matrices $(A, b, T_i^{(k+1)}, T_i^{(k)})$ are copied to the allocated global memory of the GPU. Then by specifying the number of threads and block of threads that will be used, the kernel is launched. Since the matrices are one-dimensional, the x dimension of the block can be up to the maximum number of threads per block. In this case, two values of block dimension are tested, one with blocks completely full with threads (1024 threads per block) and one with half threads (512 threads per block) to justify that full occupancy does not guarantee faster execution [54]. After the solution of the current iteration is calculated the two temperature matrices are

copied back to the host. That is needed in order to calculate the residual between the current and previous solution to test for convergence. The result of this elimination of one of the two *do*-loops, as now is executed by different threads, is that the total execution time per kernel is reduced and hence the total time required for the whole case. In Figure 3.6, the multiplication between an one-dimensional and two-dimensional matrix is schematically presented.

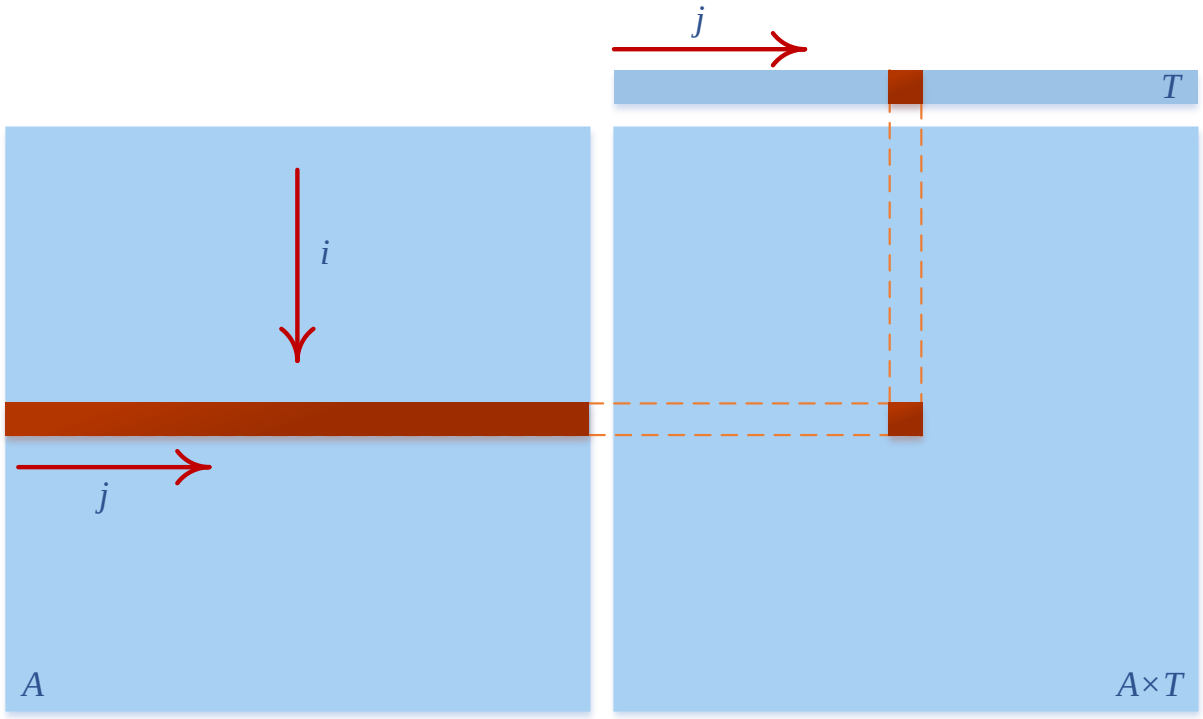


Figure 3.6: Schematic representation of the matrix multiplication operation in Jacobi method.

However, despite the total speed up of the solution, the developed kernel has a performance bottleneck. Examining the kernel structure it is understood that the *if*-statement causes threads of a warp to divert. The elimination of this problem was attempted.

3.4.3 Device version with matrix decomposition on global memory

In this version the thread divergence performance bottleneck is eliminated. In order to achieve that, the matrix that is primarily affected by the condition $i \neq j$ must be altered. As previously explained the matrix A is expressed by the diagonal matrix and the remaining residual matrix:

$$A = diagA + A'. \tag{3.20}$$

As a result, in order to eliminate the *if*-statement, it is required to decompose the matrix A, before the iterations begin. If the diagonal is filled with zeros then the total summation will not be altered. However, as seen from the Equation 3.19, the diagonal elements are needed for the division, thus it must be passed to another one-dimensional matrix. The Jacobi algorithm is the same, as described in the GPU global memory version, but now the *if*-statement is unnecessary and at the end the division with the diagonal elements takes place.

The decomposition of A is done at a separate kernel in order to save computational time. The subroutine has the name `decomposeA()` and its task is to assign the diagonal elements to another matrix and then nullify these elements. As a consequence, by eradicating the problem of thread divergence inside the kernel, less total execution time is required.

3.4.4 Device version with matrix decomposition on shared memory

Shared memory is accessed, on average, about 100 times faster than global memory (see subsection 2.3.4). Hence its usage is ideal in circumstances where certain data are accessed many time during execution inside the kernel. Inspecting the device version on

global memory, it is understood that many data are accessed more than once inside the main loop of the kernel. As a result, shared memory promises a certain speed-up of the solution.

Considering that data must first be copied from global memory to shared to be exploited multiple times, it is necessary to transfer significantly large portions to hide the memory copy latency. By constantly copying small tiles of the matrix A to shared memory and solving using now the shared memory for each tile the the kernel's execution time is reduced. In Figure 3.7 the process of copying data from matrices A, T in smaller tiles and afterwards utilizing the data directly from the tile, is schematically presented. Certain caution must be given on the indexes, as for example the tile of T matrix will have a dimension of $TileDim \times 1$ and the matrix of $nodes \times 1$.

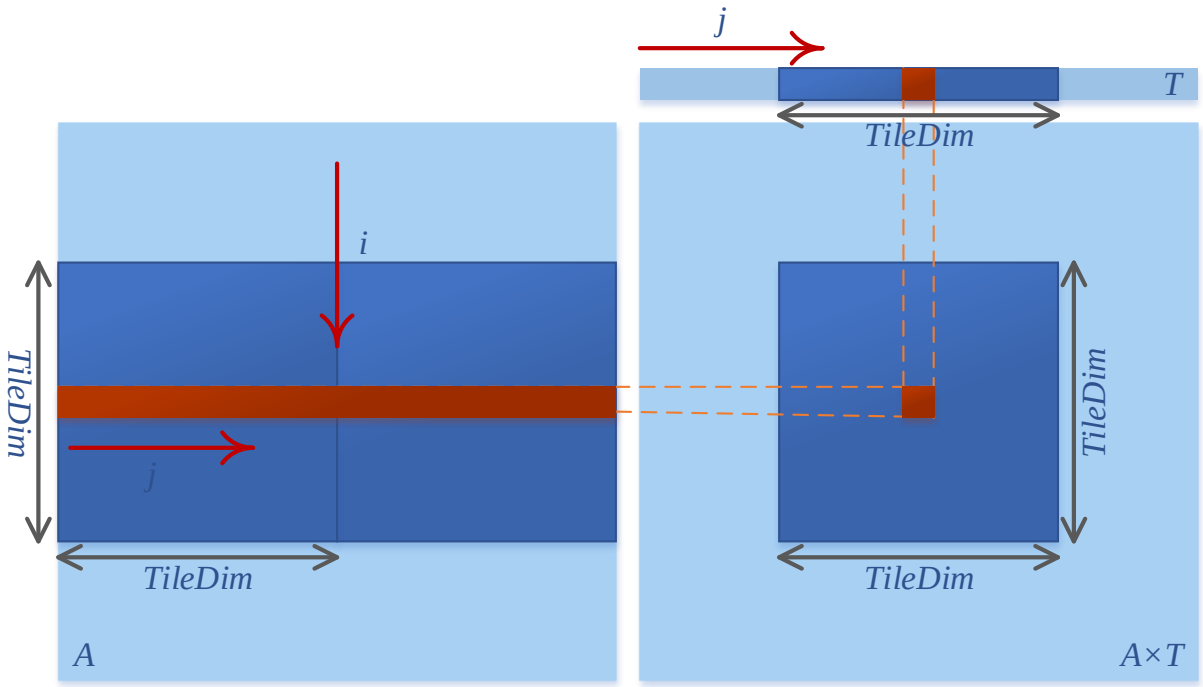


Figure 3.7: Schematic representation of the matrix multiplication operation in Jacobi method using tiles.

Unlike what may be initially expected, the use of shared memory did not introduce any performance improvement. As for this particular physical problem the number of nodes cannot easily be chosen to be multiples of `blockDim%x` (which was set to 512 threads),

inside the kernel many *if-statements* exist to fill some elements of shared memory tiles with zeros –when they exceed the total nodes, as the multiplication and subsequently the sollution will not be affected–. These branches diverge the warps and introduce a serious performance bottleneck. Additionally, more operations must be performed to fill the tiles with extra zeros. This problem was carefully examined and confronted in the problems of the next two chapters (see chapters 3,4).

3.5 Results and Discussion

The primal objective of this chapter was to solve the problem of heat conduction using parallelism techniques, in order to present the charm of CUDA. In order to measure and examine the level of parallelism, certain metrics are used as described in Section 2.6. The program execution steps are described in Apendix A.

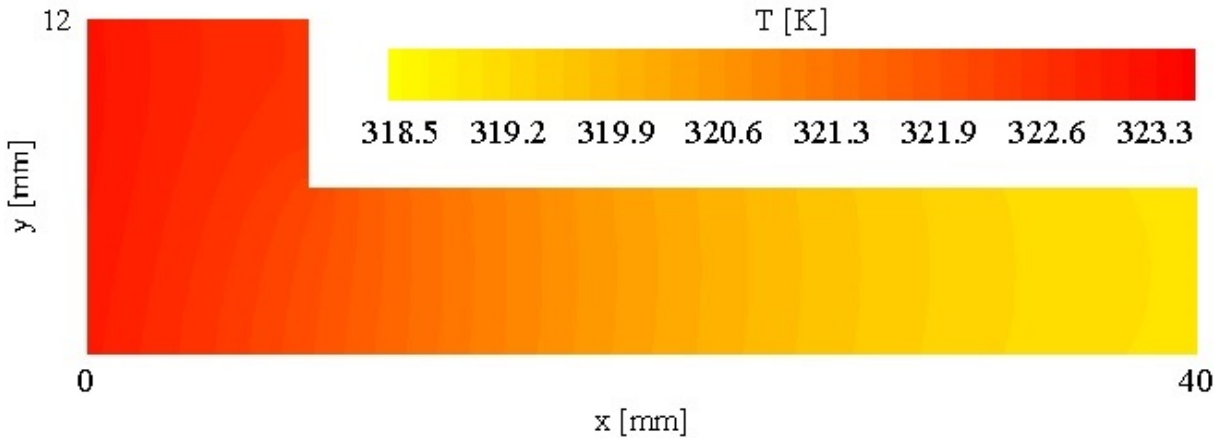


Figure 3.8: Temperature profile of the L-shaped fin from CPU version

From the contours of the two Figures 3.8 and 3.9 the correctness of the GPU results is validated, as they are completely identical with the CPU version.

The measurements for the total times of each version were done using the NVIDIA Titan Xp and NVIDIA Quadro M2200, whereas the measurements for the performance of the kernels were accomplished with the NVIDIA Quadro M2200.

Regardless the simple initial optimization techniques that were implemented, the total

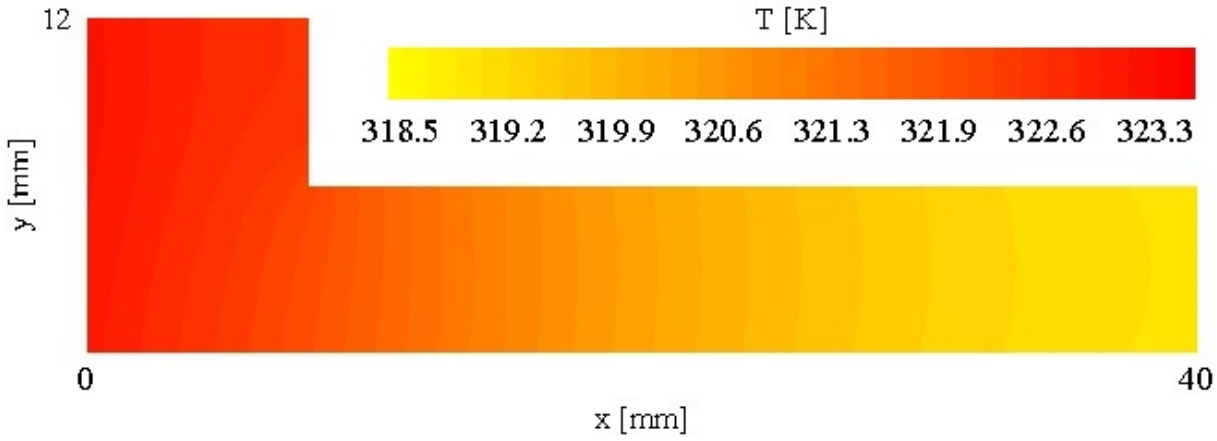


Figure 3.9: Temperature profile of the L-shaped fin from GPU version.

time was very satisfactory in comparison to CPU performance. Firstly, the program ran for different cases, but for a small accuracy of 5×10^{-5}, in order to have a fast convergence for the CPU version as well. Of course the solution was still far from the temperature profile of Figure 3.8 for such a small maximum residual, but the immense total time required from the CPU in order to converge to a solution, with stricter limits, was prohibitively huge. For the final tested case with the 29891 nodes the total time of the GPU decomposition kernel was 410 times faster than the CPU version. In Figure 3.10 the total execution time of the serial and parallel code are presented. One can notice that the NVIDIA Quadro M2200 of average performance is effectively competing the very powerful NVIDIA Titan Xp, which shows that a programmer can succeed high levels of parallelism even with average and cheap GPUs.

In Figures 3.12 and 3.13 one can see the total time execution for single and double precision data and for two different thread block sizes. The different times for single and double precision are justified by the GPU structure and the way it deals with 8 byte data. The block size has an impact on the occupancy, a very important metric to evaluate the performance of a CUDA kernel. The measurements were obtained for a total number of ten consecutive runs of the kernels, in order to obtain average values. On the figure one can see the maximum, minimum and average time of the execution time.

A question surely occurs here concerning the optimal number of threads per block.

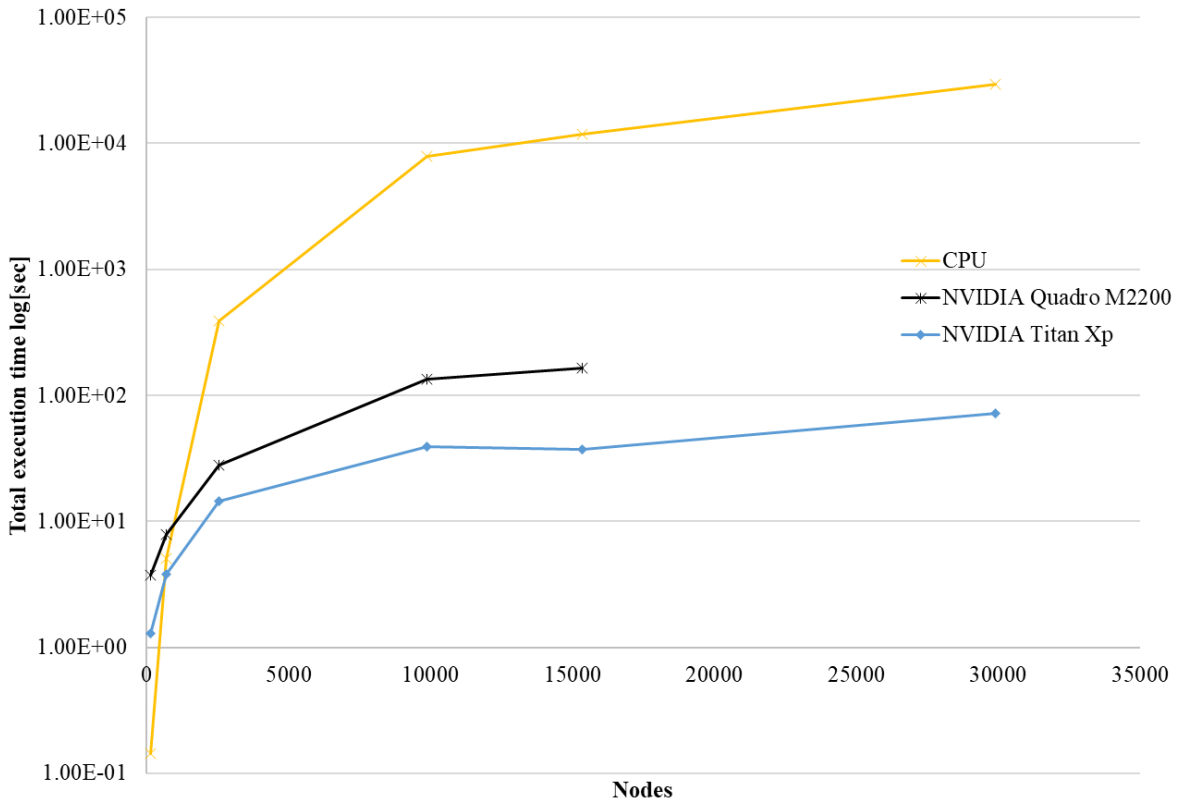


Figure 3.10: Comparison of total time execution for double precision data and block size of 512 threads between CPU and GPU versions.

As described in Section 2.6.4 an occupancy of 50% per block is the best for faster kernel execution and great utilization of the GPU capabilities. The NVIDIA Quadro M2200 for example has a maximum capacity of 1024 threads per block, thus as suggested from the literature, the optimal choice would be 512 threads per block. As seen in Figures 3.12 and 3.13 the kernel execution time for that case is slightly better than stuffing the whole block with threads, confirming the hypothesis.

A serious problem with the structure of GPU versions, that drastically affects the total execution time, is the two memory copies of the arrays `Tnew` and `T` in order to calculate the maximum residual, which take place for every iteration. As seen from Figure 3.14 for bigger arrays the time for memory copies becomes dominant. Thus the whole execution time of each version depends mainly on these data transfers from host to device and back. The construction of a kernel that calculates the maximum residual on the device, causing

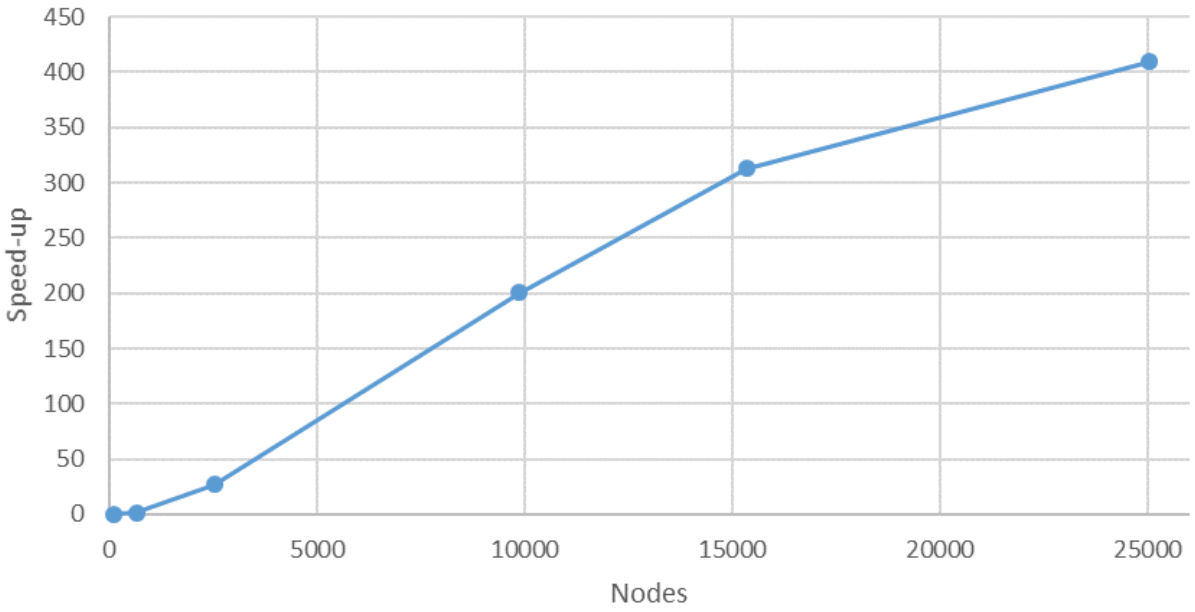
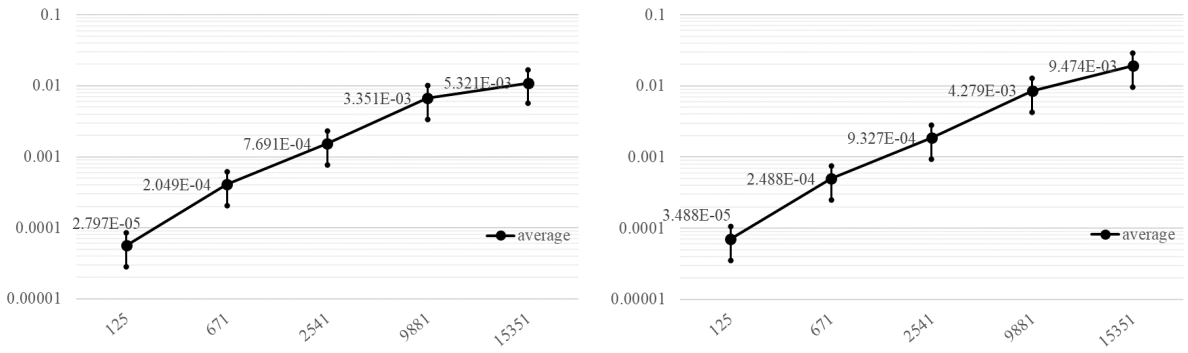


Figure 3.11: Speed-up between CPU and GPU versions.

the memory transfers between host and device to be redundant, is not greatly improving the results as normally it takes more time on GPU to perform a reduction, because the use of functions `abs()` and `max()` considerably drop down the performance of the kernel, whereas in contrary the performance is not degraded on CPU. The excessive use of host functions should be greatly considered and performed with sparingness inside kernels. Another alternative would be to use unified memory, which would completely remove the "tot mem" line from Figure 3.14.

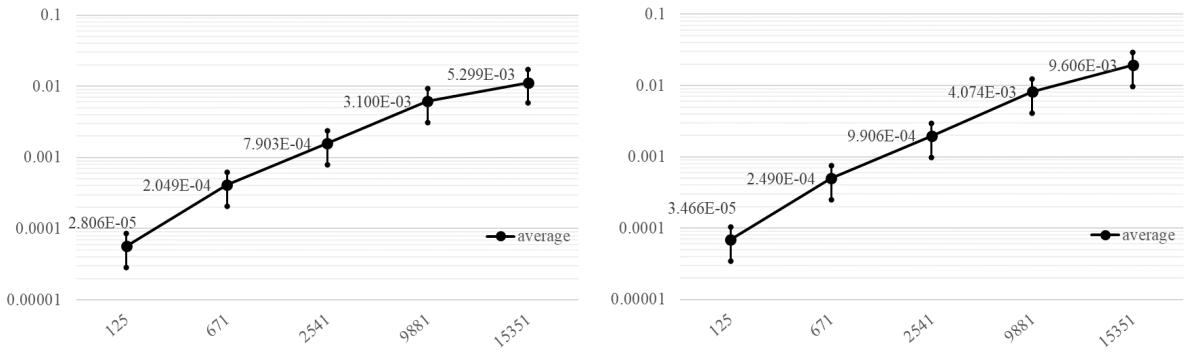
In Figure 3.15 the bandwidth that the kernels succeed is extremely good, having in mind that the theoretical bandwidth of this GPU is 88.1 Gb/s. Hence, the maximum effective bandwidth is very close to the theoretical one. Here the use of GPU's shared memory, if no thread divergence, that slows down the kernel, occured, would improve the total memory accesses. Trying to have coalesced accesses, surely improved the bandwidth and the performance in general. Running the PGI Visual profiler no global access problem is addressed. Also, allocating space directly to host's pinned memory will improve the data transfer from host to device and vise versa.

The results for the occupancy at Figure 3.16 were fairly satisfactory for the cases with



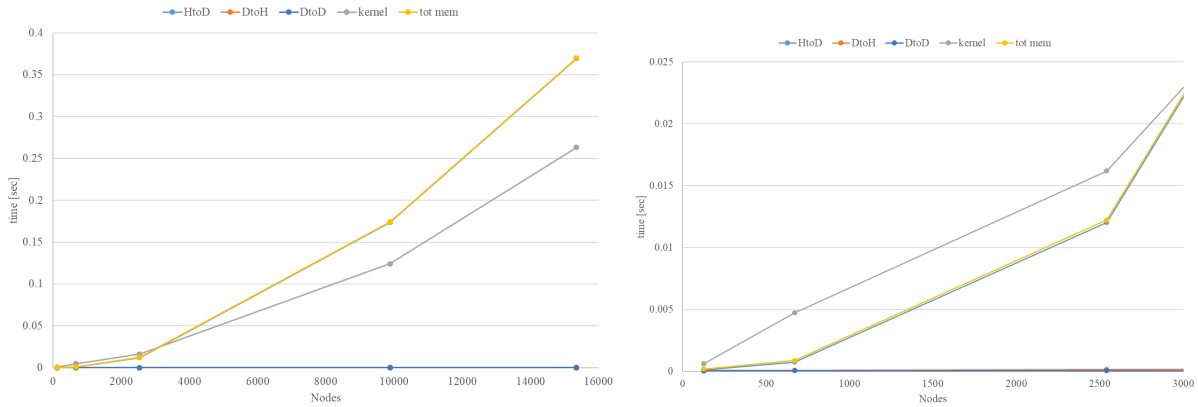
(a) (b)

Figure 3.12: Execution time for single (a) and double (b) precision data with block size of 512 threads.



(a) (b)

Figure 3.13: Execution time for single (a) and double (b) precision data with block size of 1024 threads.



(a) (b)

Figure 3.14: Time for memory copies from host to device (HtoD), from device to host (DtoH), from device to device (DtoD) and total time for memory copies as well as total kernel execution time for double precision data and block size of 512 threads.

many total nodes, whereas for the initial cases the low occupancy demonstrates that other ratios for threads per block should be used. For example in the first case a block of 512 threads is mainly empty and all threads will be forced to wait to run on a single SM. That of course does not take advantage the full capabilities of the GPU.

In Figure 3.17 the succeeded computational throughput is shown for various number of nodes. For many nodes the computational throughput is very high, as the theoretical limit of this GPU is 65.6 GFLOPs for double precision numbers, meaning that the device's capabilities are fully exploited for many nodes. The low initial computational throughput is due to the low thread percentage per block. Afterwards, as threads per case increase the GFLOPs increase considerably.

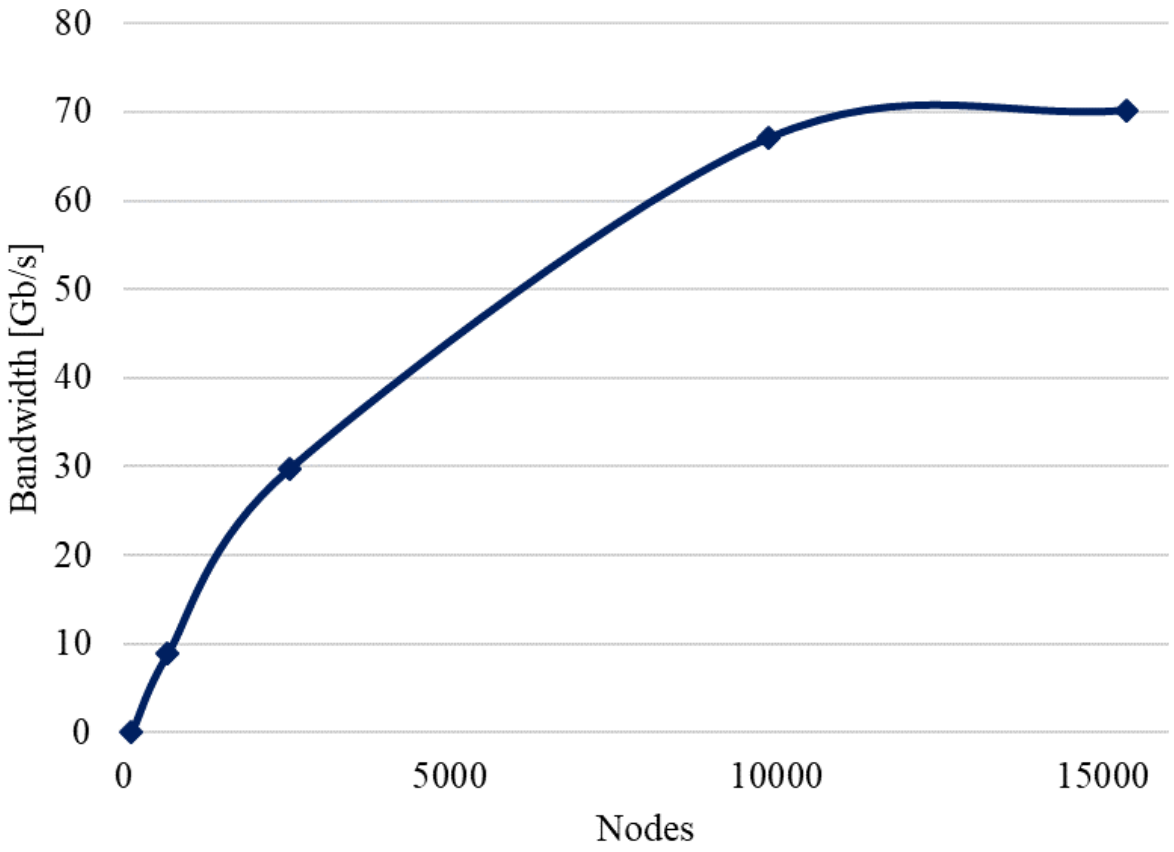


Figure 3.15: Effective bandwidth for double precision data for double precision data and block size of 512 threads.

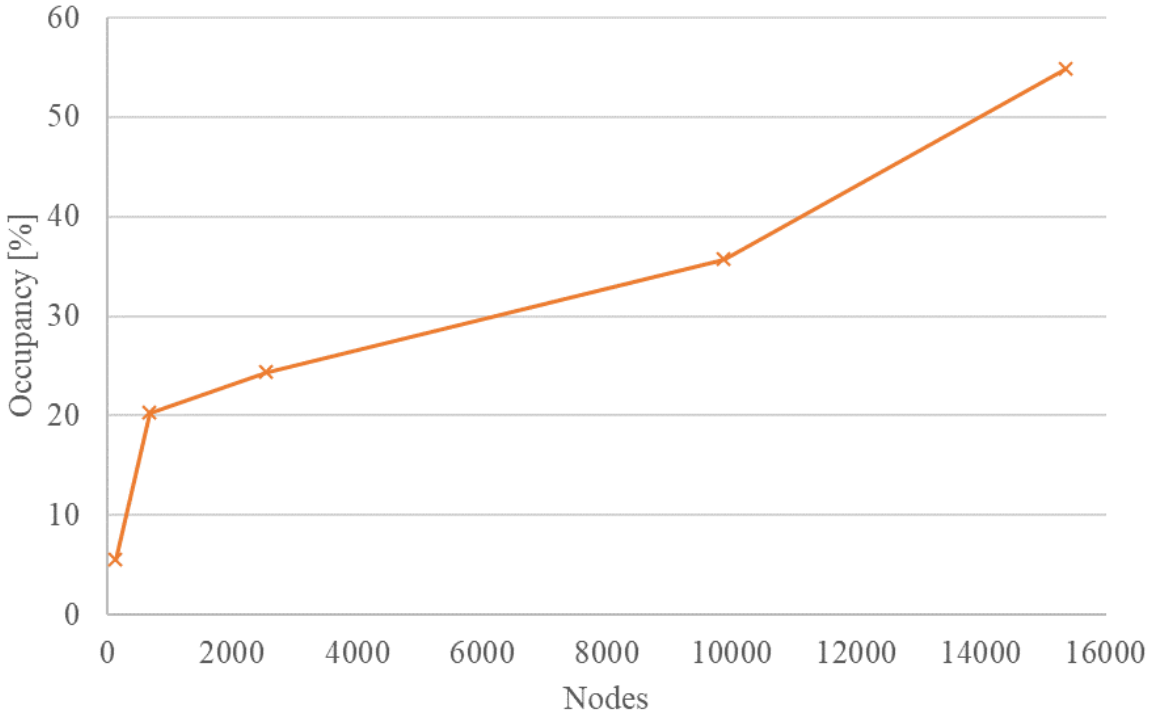


Figure 3.16: Occupancy for double precision data and block size of 512 threads

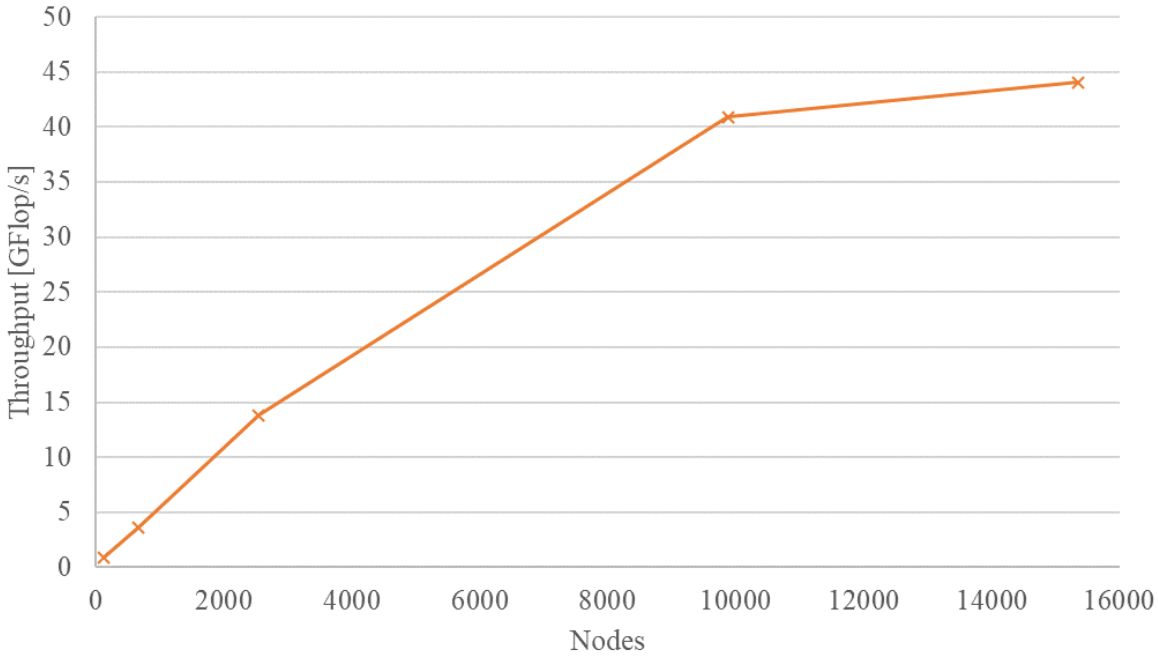


Figure 3.17: Computational throughput for double precision data and block size of 512 threads

Chapter 4

Rarefied gas flow between two parallel plates

In this chapter the flow of a monoatomic gas between two parallel plates of infinite width and length (one-dimensional flow) due to pressure gradient is examined, also known as Poiseuille flow. Practically, in applications where the flow is fully developed and two of the duct dimensions are much larger compared to the first, the problem can be reduced to the one-dimensional case.

The whole analysis to derive the necessary equations is done via the kinetic theory and specifically in the field of rarefied gases. For the collision part of the Boltzmann equation the BGK model [39] is used and for the discritization of the microscopic velocities the deterministic Discrete Velocity Method (DVM) [62].

4.1 Flow configuration and kinetic formulation

In Figure 4.1 a schematic representation of the problem is shown. The gas flows between two parallel plates, due to a pressure difference on the sides and towards the smallest pressure. The temperature throughout the whole domain has the same value. As this pressure gradient is not of significant magnitude, the flow is linear and is known as Poiseuille flow. The steady one-dimensional fully developed flow between two parallel

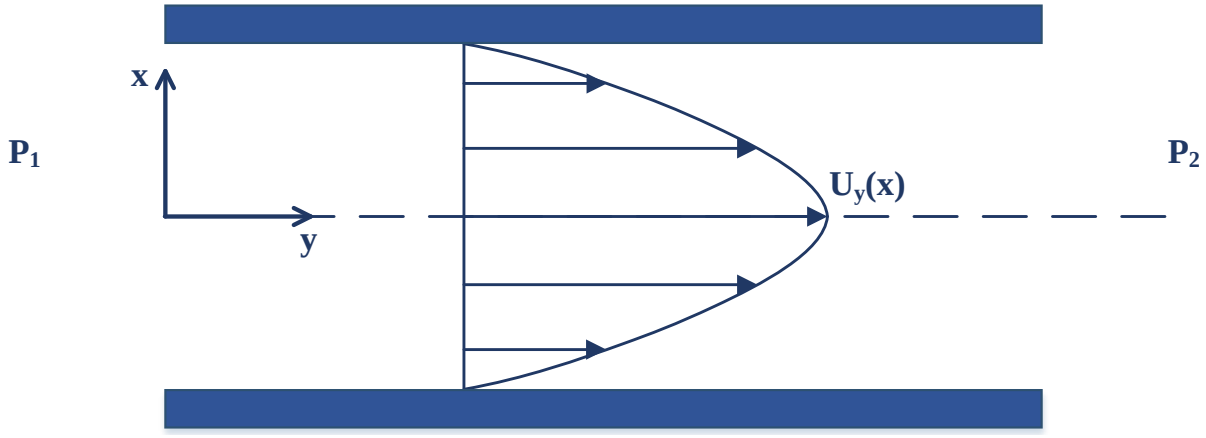


Figure 4.1: one-dimensional Poiseuille flow.

plates due to pressure gradient can be solved using the linearized equation according to the BGK model. The governing non-dimensional BGK equation is:

$$\mu \frac{\partial Y(x,\mu)^{(k+1/2)}}{\partial x} + \delta Y(x,\mu)^{(k+1/2)} = \delta u(x)^{(k)} + \frac{1}{2}, \tag{4.1}$$

with the macroscopic velocity described as:

$$u(x)^{(k+1)} = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{+\infty} Y(x,\mu)^{(k+1/2)} e^{-\mu^2} d\mu. \tag{4.2}$$

At the boundaries the distribution function is assumed to be zero, thus the boundary conditions are:

- $Y(-1/2,\mu) = 0,\ \mu > 0$
- $Y(1/2,\mu) = 0,\ \mu < 0$

The parameters in Equations 4.1 and 4.2 are the spatial variable $x\epsilon[-1/2, 1/2]$, the x-component of the molecular velocity $\mu\epsilon(-\infty,\infty)$, the unknown distribution function $Y = Y(x,\mu)$ and the macroscopic velocity $u = u(x)$. The variable k denotes the iteration index.

The non-dimensional flow rate is found as an integral on the velocity profile and is given according to the following equation:

$$G = 2 \int_{-1/2}^{1/2} u(x)dx. \tag{4.3}$$

4.2 Serial code

In order to obtain the solution an iterative procedure is required till convergence takes place. The macroscopic velocity $u(x)$ is assumed at the right hand side of Equation 4.1 and consequently the distribution function $Y(x, \mu)$ can be determined. After $Y(x, \mu)$ is calculated, the new value is substituted at the right side of Equation 4.2 and thus the new $u(x)$ can now be computed. The new value of macroscopic velocity is compared with the previous one, to check for convergence. The iterations continue till convergence is reduced below a minimum tolerance. This iterative procedure corresponds to an outer *do-loop* that encloses the discretized versions of Equations 4.1 and 4.2.

A set of discrete molecular velocities p_m, with $m = 1, 2, ..., M$, is defined at the beginning of the program (or `CC(I)` in the code script). In this case, the arithmetic scheme that was chosen is the Gauss-Legendre and each root of the Legendre polynomials represents one molecular wight. The number of roots and the corresponding weights is determined as an input from the user. Half of the pairs –nodes and weights– need to be administered as the opposites are the same values but negative, hence this is done independently inside the program.

```
1: open(1, file ='g128.dat', status ="old")
2: do i = 1, M / 2
3:     read(1, *) CC(i), WW(i)       ! Positive velocities
4:     CC(i + M / 2) = - CC(i)       ! Negative velocities =positive*(-1)
5:     WW(i + M / 2) = WW(i)         ! Weight factors
6: end do
```

Listing 4.2.1: Reading Gauss-Legendre roots and weights.

The file name corresponds to the total number of Gauss-Legendre roots (both positive and negative) that will be read and eventually used during execution. Only half values –roots and weights– will be used, thus despite the file name's description, only half values should be found inside this ".dat" file, where the negatives are created inside the *do-loop* of the code. Altering this file name on the host code, various number of Gauss-Legendre roots and weights can be inputted in the code.

A transformation must necessarily occur so that the discreet molecular velocities will belong to the field $[0,\infty)$ rather than $[-1,1]$. This can easily be accomplished by applying a new variable c as in Equation 4.4. It is important to perform this step as later on it will be proved that the derived equations are of the exact same structure. As a result only the discrete set of $c \in [0,\infty)$ should be used.

$$c = \frac{1+p}{1-p}, \qquad dc = \frac{2dp}{(1-p)^2}. \tag{4.4}$$

The above new variable c introduces a a set of discreet velocities c_m, $m = 1,2,...,M$, in $[0,\infty)$.

```
1: do i = 1, M
2:     C(i) = (1.0_fpKind + CC(i)) / (1.0_fpKind - CC(i))
3:     W(i) = 2.0_fpKind * WW(i) / (1.0_fpKind - CC(i)) ** 2.0_fpKind
4: end do
```

Listing 4.2.2: Transformation of discrete velocities and weighting factors for the integration.

Afterwards, the spatial domain is divided into I equally spaced intervals with length $h = 1/I$ with a midpoint i for each interval. In Figure 4.2 the spatial grid is schematically presented. Assuming the distribution function $Y(x,\mu)\Big|_{i,m}$ is equal to $Y(x_i,\mu_m) = Y_{i,m}$

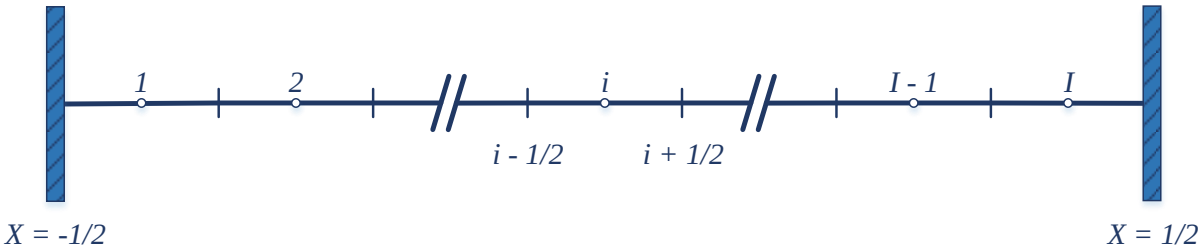


Figure 4.2: Spatial grid of one-dimensional Poiseuille flow.

and discretizing the kinetic equation at nodes (i,m), where $i = 1,2,...,I$, $m = 1,2,...,M$ one obtains Equation 4.5.

$$\mu_m \frac{\partial Y(x,\mu)^{(k+1/2)}}{\partial x}\bigg|_{i,m} + \delta Y(x,\mu)^{(k+1/2)}\Big|_{i,m} = \delta u(x)^{(k)}\Big|_{i} - \frac{1}{2}. \tag{4.5}$$

Both terms of the left hand of Equation 4.5 can be approximated using Taylor series as follows:

$$\frac{\partial Y}{\partial x}\bigg|_{i,m} = \frac{1}{h}\left(Y_{i+\frac{1}{2},m} - Y_{i-\frac{1}{2},m}\right) + O\left[h^2\right], \tag{4.6}$$

$$Y_{i,m} = \frac{1}{2}\left(Y_{i+\frac{1}{2},m} + Y_{i-\frac{1}{2},m}\right) + O\left[h^2\right]. \tag{4.7}$$

The result is to acquire a system of Equations 4.8 that is solved by following the particle trajectories.

$$\mu_m \frac{Y^{(k+1/2)}_{i+\frac{1}{2},m} - Y^{(k+1/2)}_{i-\frac{1}{2},m}}{h} + \frac{\delta}{2}\left(Y^{(k+1/2)}_{i+\frac{1}{2},m} + Y^{(k-1/2)}_{i+\frac{1}{2},m}\right) = \frac{\delta}{2}\left(u_{i+\frac{1}{2}} + u_{i-\frac{1}{2}}\right)^{(k)} + \frac{1}{2}. \tag{4.8}$$

By introducing the parameter $T_0 = \dfrac{h\delta}{2\mu_m}$ the above system of Equations 4.8 can be separated to two other systems; one for each positive or negative value of the discretized variable c_m:

- For $\mu_m > 0$:

$$Y^{(k+1/2)}_{i+\frac{1}{2},m} = [1+T_0]^{-1}\left\{(1-T_0)\,Y^{(k+1/2)}_{i-\frac{1}{2},m} + T_0\left(u_{i+\frac{1}{2}} + u_{i-\frac{1}{2}}\right)^{(k)} + \frac{h}{2\mu_m}\right\}, \tag{4.9}$$

  where $m = 1, 2, ..., M$ and $i = 1, 2, ..., I$ and a boundary condition $Y_{\frac{1}{2},m} = 0$.

  In order to reduce the total run time, some variables that are reused many times during execution are calculated outside of the *do-loop*, at the beginning. This step, reduces the run time substantially, as fewer operations are performed. For instance, running the serial code for 64 microscopic velocities, 4097 nodes and 1535 iterations the total time decreases from 7.587 to 4.138 seconds.

```
1: T0 = 0.5_fpKind * H * delta / C(j)
2: T01 = T0 / delta
3: l1 = (1.0_fpKind - T0) / (1.0_fpKind + T0)
4: l2 = T0 / (1.0_fpKind + T0)
5: l3 = T01 / (1.0_fpKind + T0)
6:
```

```
!Boundary Conditions
Y(1, j) = 0.0_fpKind  ! Left wall
do i = 2, N
    Y(i, j) = (l1 * Y(i - 1, j) + l2 * (u(i - 1) + u(i)) + l3)
end do
```

Listing 4.2.3: Calculation of distribution function Y for positive velocities

- For $\mu_m < 0$:

$$Y_{i-\frac{1}{2},m}^{(k+1/2)} = [1-T_0]^{-1}\left\{(1+T_0)\,Y_{i+\frac{1}{2},m}^{(k+1/2)} - T_0\left(u_{i+\frac{1}{2}}+u_{i-\frac{1}{2}}\right)^{(k)} - \frac{1}{2\mu_m}\right\}, \quad (4.10)$$

where $m = 1,2,...,M$ and $i = I, I-1, ...,1$ and a boundary condition $Y_{I+\frac{1}{2},m} = 0$.

```
!Boundary Conditions
Y(N, j + M) = 0.0_fpKind ! Right wall
do i = N - 1, 1, - 1
    Y(i, j + M) = (l1 * Y(i + 1, j + M) + l2 * (u(i) + u(i + 1))  &
        + l3)
end do
```

Listing 4.2.4: Calculation of distribution function Y for negative velocities

In Equation (4.10), if one defines $T_0 = \dfrac{h\delta}{2|\mu_m|}$, then it can be rewritten as:

$$Y_{i-\frac{1}{2},m}^{(k+1/2)} = [1+T_0]^{-1}\left\{(1-T_0)\,Y_{i+\frac{1}{2},m}^{(k+1/2)} + T_0\left(u_{i+\frac{1}{2}}+u_{i-\frac{1}{2}}\right)^{(k)} + \frac{1}{2|\mu_m|}\right\}. \quad (4.11)$$

Expressions 4.9 and 4.11 have the same form, thus as previously mentioned the set of new discrete velocities c_m must belong to $[0,\infty)$. The only difference corresponds to solving during execution, where in Equations 4.9 the solution is obtained by moving from

bottom to top (following coordinates of Figure 4.1) and in Equations 4.11 from top to bottom. Ergo two *do-loops* exist –one for positive and one for negative values of c_m– to solve for the calculation of the distribution function, which are subsequently enclosed into another *do-loop* to compute the molecular velocities.

After the calculation of the distribution function, the macroscopic velocities u are computed, by integrating over c. The integration happens for $c \in [0, \infty)$ and this is another reason why only these values are needed at the beginning during the transformation of Legendre roots from $[-1, 1]$.

$$
\begin{aligned}
u(x)^{(k+1)} &= \frac{1}{\sqrt{\pi}} \int_{-\infty}^{+\infty} Y(x,\mu)^{(k+1/2)} e^{-\mu^2} d\mu \\
&= \frac{1}{\sqrt{\pi}} \int_{0}^{+\infty} Y(x,\mu)^{(k+1/2)} e^{-\mu^2} d\mu + \frac{1}{\sqrt{\pi}} \int_{0}^{+\infty} Y(x,-\mu)^{(k+1/2)} e^{-\mu^2} d\mu \quad . \qquad (4.12) \\
&= 2 \left[\sum_{m=1}^{M} Y(x,\mu_m)^{(k+1/2)} w_m e^{-\mu_m^2} + \sum_{m=M}^{2M} Y(x,\mu_m)^{(k+1/2)} w_m e^{-\mu_m^2} \right]
\end{aligned}
$$

```
1: do i = 1, N
2:     u(i) = 0.0_fpKind
3:     do j = 1, M
4:         u(i) = u(i) + W(j) * (Y(i, j) + Y(i, j + M)) * EXP(- C(j) ** 2 &
           .0_fpKind)
5:     end do
6:     u(i) = u(i) / RPi
7: end do
```

Listing 4.2.5: Calculation of macroscopic or bulk velocity.

At last, after the solution converges to a value, as a post-process, the flow rate G can

be calculated. In this case the trapezoidal rule is used [63].

$$G = 2\int_{1/2}^{-1/2} u(x)dx = h\left[u_1 + 2\sum_{i=2}^{N-1} u_i + u_N\right]. \tag{4.13}$$

```
1: S = 0
2: do i = 2, N - 1
3:     S = S + u(i)
4: end do
5: flowRate = (u(1) + 2.0_fpKind * S + u(N)) * H
```

Listing 4.2.6: Calculation of flow rate

4.3 CUDA implementation

There are two main parts of the serial code that need to be parallelized. These are the calculations of the distribution function and the macroscopic velocity along the domain. Subsequently, the relevant error between two iterations can also constitute a separate kernel, not only for faster execution, but to eliminate the need of any array transfers between host and device during solving, inside the outer *do-loop* for convergence.

4.3.1 Kernel of distribution function

As described previously the calculation of the distribution function occurs starting from the side wall and moving towards the center. Thus, the parallelization will be implemented on the microscopic velocities M. The basic concept is that each thread is responsible to solve for one microscopic velocity. As a result, the calculation of every Y will be performed simultaneously. Moreover, to further increase the parallelism, 1 block is assigned for the positive velocities and another one for the negative. In Figure 4.3 the concept of parallelization is schematically presented. It is important to be noted that for

this specific physical problem no more than 1024, which is the threads per block limit, microscopic velocities will be used, as 512 is more than enough to obtain an accurate solution.

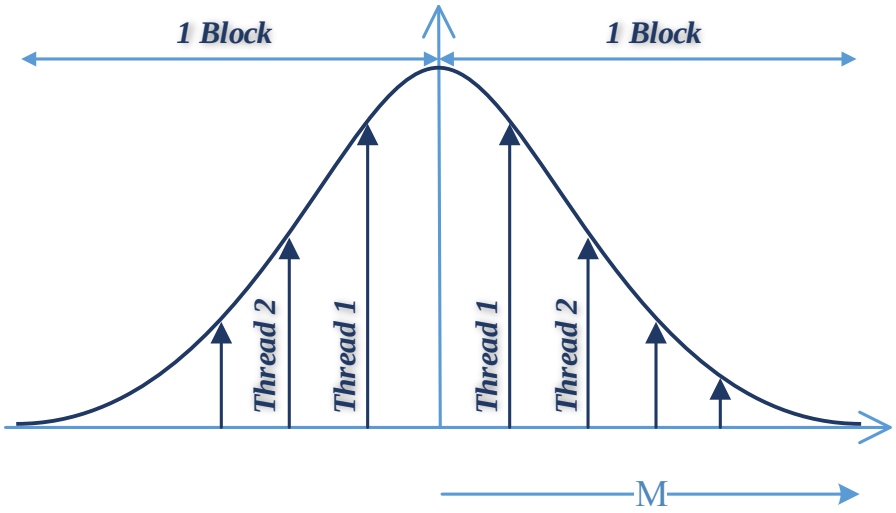


Figure 4.3: Schematic representation of the parallelization of distribution function.

Various version were developed as the different memories, that a GPU is equipped with, vary in performance. According to the memory utilized, each version is named relevantly. It is stated that constant memory was used for all versions, for the distance of two nodes H and the rarefaction parameter, meaning the variables `H_c, delta_c` respectively.

4.3.1.1 Global memory

Following the basic parallelism concept, this kernel was initially constructed, using exclusively the global memory. It is essential to state that this kernel, for reasons that are later explained, was formed by direct application of Equations 4.9 and 4.11 and not the shortened ones, with some variables calculated outside the *do-loop*, as presented in the serial code, in order to reduce the number of performed operations inside the *do-loop* and subsequently the run time. Therefore, the parallel results are expected to be inefficient and inadequate to serial execution.

Considering that even in the most demanding cases the solving of the one-dimensional Poiseuille flow does not require many molecular velocities, the kernel is launched with M threads and 2 blocks. After the operation `T0 = H_c * delta_c / C(j)` is performed at the beginning of the kernels, threads that belong to the first block are assigned to solve for the positive microscopic velocities and those of the second block for the negative ones. The separation of the threads happens with an *if-else* statement.

Clearly from a GPU performance perspective it is optimal to run for threads multiples of 32 (one warp) for compute capability 2.0 and higher, thus the Gauss-Legendre roots and weights should be chosen accordingly to be divisible by 32. It is preferential to select more microscopic velocities than needed and satisfy this condition, than using the exact quantity the user has in mind, because in the latter case the performance will not be optimal.

4.3.1.2 Global memory with parameters directly residing on registers

Observing Equations 4.9 and 4.11, it is understood that too many accesses on global memory take place inside the *do-loop* for the spatial grid. These accesses are extremely slow and eventually hinder the whole execution. The reduction of global memory accesses is essential especially as the number of spatial nodes increases.

In Section 2.5 is mentioned that registers is the fastest on-cache memory and its usage is vital to reduce kernel's run time. It is also mentioned that dynamically accessed arrays do not reside on registers but on local memory, which basically is another way of accessing global memory, with much slower bandwidth compared to registers. Thus `C(j)` is one example of an array that will not be moved to registers and therefore when used inside the *do-loop* it will result in a performance bottleneck.

Moreover, both equations include many parameters that could be substituted by fewer. Example in that direction is the coefficient of the parenthesis of macroscopic velocities in the right hand of Equation 4.9 `T0/(1-T0)`. Also, reducing the number of performed operations, increases the computational throughput of the kernel, which is beneficial in the overall performance. In the given example, not many variables exist to surpass the

capacity limit of registers or many copies from global memories, however during execution the operation to calculate the term $T0/(1-T0)$ will happen unnecessarily several times inside the *do-loop*.

The solution is to reduce and substitute as much as possible the variables that exist in Equation 4.9 by fewer ones and load the latter at the beginning of the kernel. This act will reduce the number of operations performed during kernel execution and will load array `C(j)` on registers. The same optimization was followed to reduce the execution time of the serial code.

4.3.1.3 Texture memory with parameters directly residing on registers

Texture memory is ideal for uncoalesced accesses to global memory and for constant arrays. The calculation of distribution function necessarily needs the macroscopic velocities `u` computed in the previous iteration. Therefore, the array of `u` is kept constant during the solution of `Y` and can be committed to texture memory. Additionally, the array of discrete molecular velocities `c` is also constant and can be loaded in a similar way.

At the beginning of the host code and outside the main *while-loop* pointers assign the targets `u` and `c` arrays to `uTex` and `cTex` arrays, respectively, which are device arrays residing on texture memory. Now the names `uTex` and `cTex` are used inside the kernel.

4.3.1.4 Shared memory with parameters directly residing on registers

Excluding thread-private memories, shared memory is the fastest one, thus its usage is important and especially when some values are utilized many times throughout execution. Almost all elements of the `u` array are used twice for the calculation of a single value of `Y`, therefore shared memory for the macroscopic velocities promises some improvement. The usage of shared memory comes in two different versions.

On one hand, if the total number of elements of u, that is the spatial nodes, does not exceed the capacity of shared memory then all values can be loaded there at the beginning of the kernel. The total capacity of shared memory on NVIDIA Quadro M2200 GPU is 49152 bytes, or 48 kB. If launching the kernel with double precision (8 bytes) then the

maximum number of nodes that can be used is $49152/8 = 6144$ nodes. It is remarked that this number of nodes is more than enough for this specific problem, thus the usage of this kernel does not arise any difficulties.

On the other hand, if more than 6144 nodes are used then the method of tiles can be implemented. During this method, the one-dimensional array of macroscopic velocities is divided into several tiles, which are subsequently loaded with two values of u. In order to fill those tiles the M threads are used. This version was kept at last, although slightly slower, due to its increased flexibility.

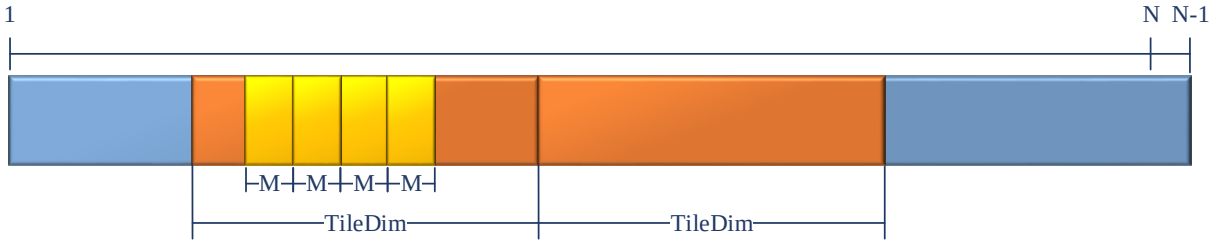


Figure 4.4: Introduced conditions with final version.

Both versions with shared memory arise one limitation that should be met, since the array u is loaded into several equally sized tiles, that the size u must be perfectly divisible by the dimension of the tiles. Moreover, for the second version, another limitation is that since the tile is loaded using the M threads, the dimension of a tile must be perfectly divisible by M. It should be noted that these limitations are more of conditions that should be met as they do not really limit the user. Several steps were made to eliminate these two prerequisites, like filling the last tile with zeros, in case no more nodes exist, and using two dimensional blocks to fill tiles of arbitrary size, however all cases were proven slower. That means that it is advantageous to use much more nodes and microscopic velocities than the ones actually needed in order to fulfill these conditions, rather than trying to eliminate them. In Figure 4.4 the two conditions are schematically presented.

Shared memory loads two elements of macroscopic velocities at a time. Examining the form of the shared memory loads it can be seen that $nodes - 2$ elements will be loaded twice from global memory to two different shared memory banks. For example global memory bank for `u(2)` will be accessed by thread 2 and thread 3 simultaneously. This

generates the problem that global memory accesses are not coalesced anymore as two threads may ask for access on the same global memory bank. One way to eliminate this problem is to access u through texture memory. Accomplishing that results in negligible improvement even for many nodes and for reasons of simplicity it is not preserved.

4.3.2 Kernel of macroscopic velocities

The calculation of macroscopic velocities takes substantially less time than of distribution function. However, creating a kernel will not only speed up the process, but also will prevent the need to transfer both arrays `Y` and `u` from device to host and vice versa. Memory transfers when the matrices are exceptionally large may introduce a serious performance bottleneck.

The kernel is launched with N blocks and M threads per block. Each thread is responsible to solve for one microscopic velocity and each block is designated to one spatial node. The main body of the kernel is a reduction scheme, which is thoroughly described in Appendix C, as a total sum must be obtained, in order for all microscopic velocities to find the macroscopic velocity of each spatial node, i.e. for each block. Another version that was tested, instead of the reduction scheme, was to launch the kernel with 512 threads per blocks and adequate blocks to fill $M \times N$ elements and execute a simple *do-loop* inside to find `u`. The indices were conformed accordingly. This was proven to be approximately one third slower than the previous kernel structure, thus it was quickly rejected. Out of much experimentation (ex. using various two-dimensional blocks) this particular structure with reduction was proven the fastest.

To increase kernel's performance, shared memory was used. Initially, shared memory was loaded with M elements, which are given by first term of the right hand side of Equation 4.12. This offered a huge improvement in the performance of the previous version, as observed later by the results. As a final version, to further reduce run time, the tile of shared memory is loaded with two elements, as in Equation 4.12, performing both operations, hence the kernel needs to be launched with $M/2$ threads per block.

A condition that is introduced by the last two versions with shared memory utilization

is that M must be the results of a power of two. The reduction with sequential addressing works for specific number of elements, since the index of the array is constantly divided by a factor of 2. It is stated here that similarly to the previous kernel of distribution function eliminating this particular limitation, by partially filling the last tile with zeros and introducing an if statement in the main body the performance is greatly hindered and is actually more time consuming than choosing another set of microscopic velocities that satisfies this condition.

4.3.3 Structure of complete parallel code

The complete code, with both kernels is schematically presented in Figure 4.5. The total improvement is offered with the simultaneous execution for negative and positive velocities, by thread blocks, and for each microscopic velocities, by threads per block, as well as the simultaneous calculation of macroscopic velocities. Additionally, the maximum residual is also calculated inside a kernel offering another enhancement in the performance of the total code.

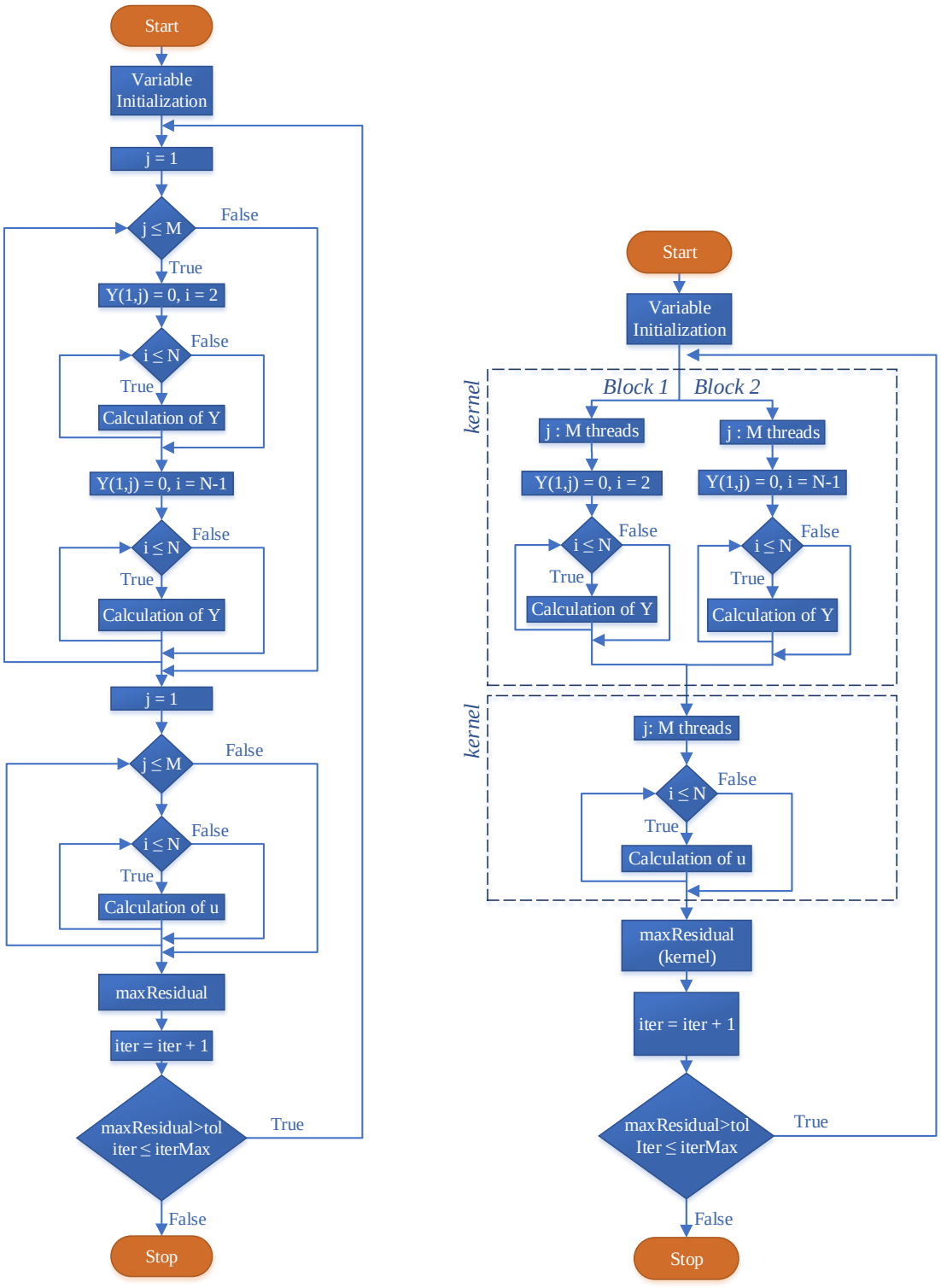


Figure 4.5: Schematic representation of the complete code of host and device versions.

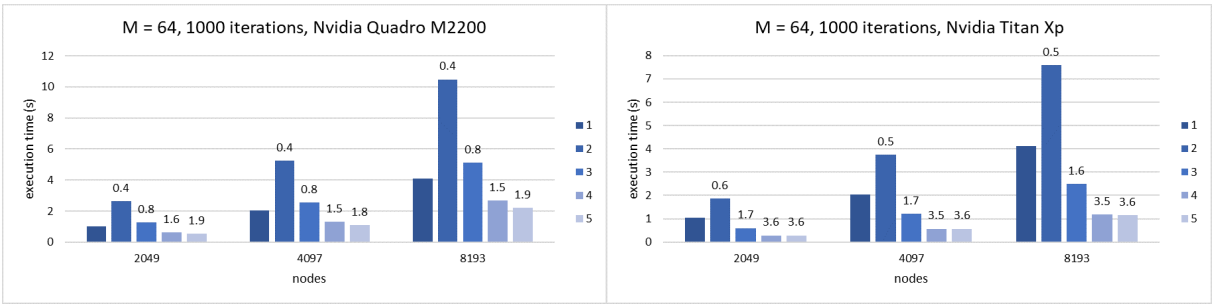
4.4 Results and discussion

Time measurements and metrics of performance were obtained for each kernel separately and run times from the complete code for various deltas. For the kernel calculating the distribution function all different cases were examined. These are the following: version 1 is the serial code with altered equations, as explained in Section 4.3.1.2, version 2 is the parallel device code on global memory, version 3 is the parallel code on global memory with altered equations, version 4 is the parallel code on global memory with altered equations and u, C on texture memory and version 5 is the parallel code with altered equations on shared memory with tiles. The program execution steps are described in Apendix A.

The results were obtained for $M = 64, 128, 512$ and $N = 2049, 4097, 8183$. The execution time of all versions was measured for 1000 iterations and on both GPUs, NVIDIA Quadro M2200 and NVIDIA Titan Xp.

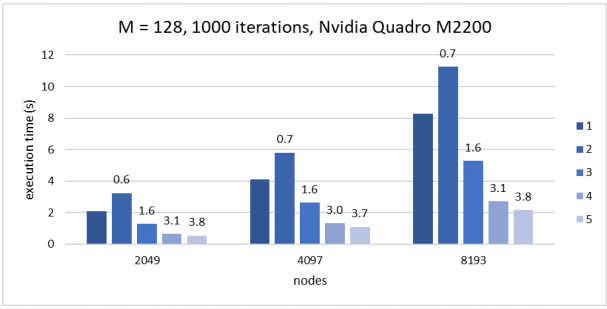
On top of the bars in Figure 4.6 the speed-up of the specific version is shown. The shared memory version seems to offer highly reduced run times, a phenomenon which is improved as the number of threads per block increases. As aforementioned, version 2 does not include the altered equations, which version 1 and other have, thus the total execution time and speed-up are worse than the CPU time. However, this version was included in the graphs to show the importance of utilizing the registers of the GPU and how much the performance is improved by this extremely fast memory.

Moreover, further observation of the figures shows that the speed-up of each version is not affected by the total number of spatial nodes. This is positive as kernel's performance does not drop when the *do-loop* inside the kernel performs more iterations. Increasing the threads per block boosts the succeeded speed-up of every version. Finally, another important conclusion is that even by using GPU cards not designed for computations, such as the NVIDIA Quadro M2200 the results are extremely satisfactory and one might say comparable to the results from powerful GPUs, such as the NVIDIA Titan Xp. This confirms that CUDA offers a very efficient alternative to parallel computing at an ex-

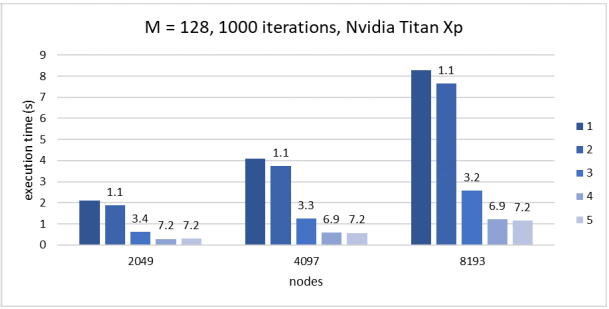


(a)

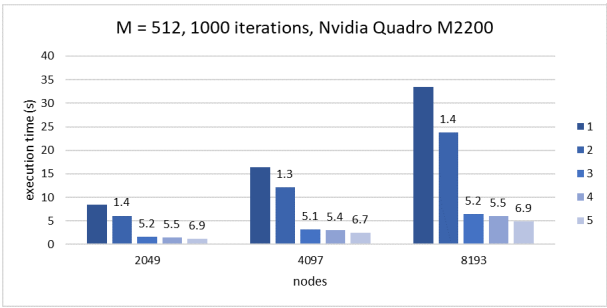
(b)



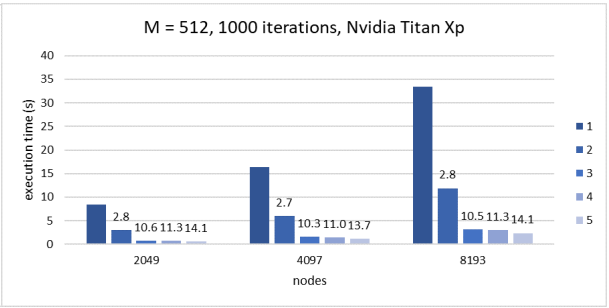
(c)



(d)



(e)



(f)

Figure 4.6: Results for kernel of distribution function.

tremely, non-comparable low cost. Therefore, even with a mobile GPU the programmer can accelerate his code sufficiently and obtain relatively fast results. In Table 4.1 execution time of serial code and speed up are presented, only for 4097 nodes as similar behavior is observed for the other examined number of spatial nodes.

M	CPU	Nvidia Quadro M2200 speed-up	Nvidia Titan Xp speed-up
64	2.034 s	1.8	3.6
128	4.103 s	3.7	7.2
512	16.343 s	6.7	13.7

Table 4.1: Results for kernel of distribution function for 4097 nodes.

Memory bandwidth and computational throughput metrics were obtained on the NVIDIA Quadro M2200 GPU, which are presented in Figures 4.7 and 4.8. Both figures have the curves that was expected. Memory bandwidth is asymptotically increased till a certain maximum limit is met. Moreover the increasing trend of GFLOPs reaches an upper limit as the number of threads grows. The maximum results are far from the device's limits of $88Gb/s$ and 65.6 GFLOPs.

Running the PGI Visual Profiler for this kernel, it shows that no warp divergence or bank conflicts exist. Although, a problem related to global access pattern appears. Compiling the code with `-Mcuda=lineinfo` flag, the Profiler shows that a memory coalescing problem occurs when loading two values of macroscopic velocities simultaneously. Solving this problem by using texture memory, an option which is ideal for uncoalesced accesses to global memory, offers a negligible improvement of the total execution time, specifically by few milliseconds for 1000 iterations, therefore it was not adapted to the final version. This shows that even if a performance bottleneck exists in the code, sometimes it may not affect performance to the slightest.

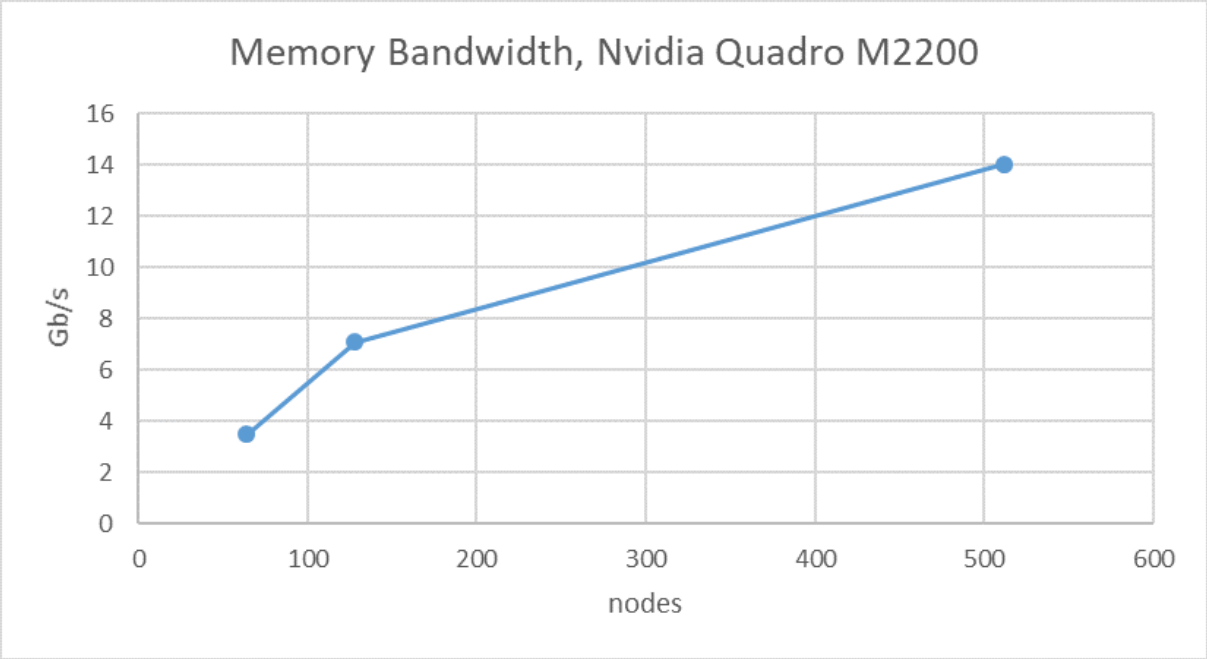


Figure 4.7: Memory bandwidth of kernel of distribution function.

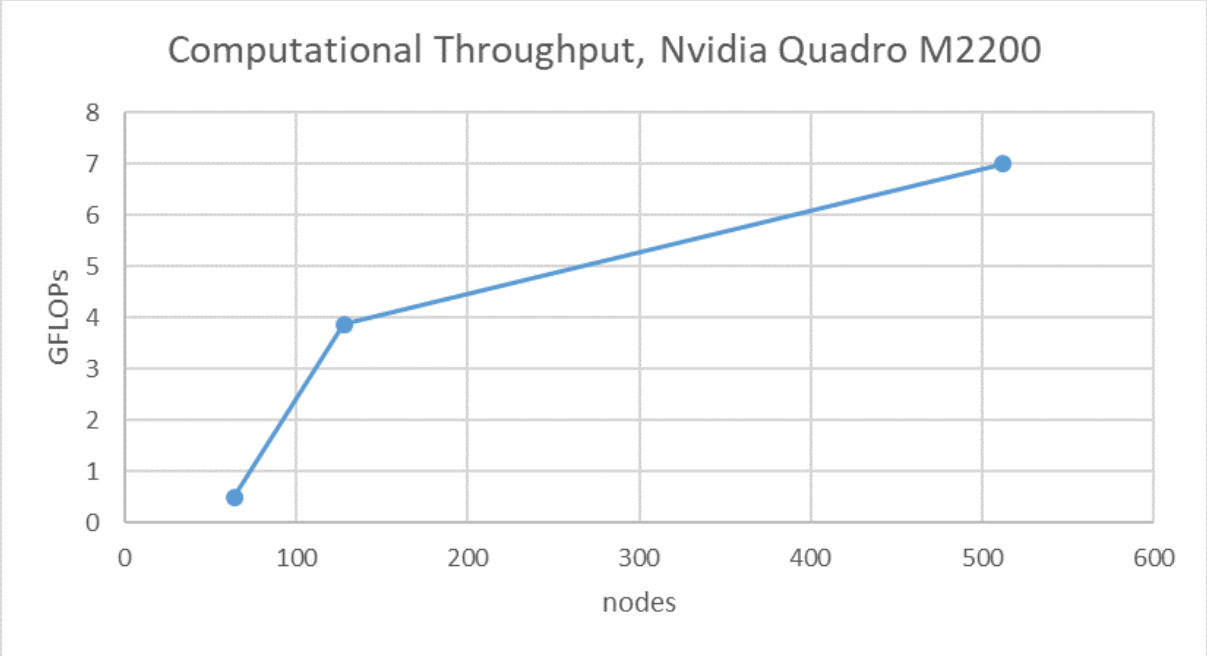


Figure 4.8: Computational throughput of kernel of distribution function.

For the kernel calculating the macroscopic velocities the versions that were examined were the following: version 1 is the serial code, version 2 is the parallel code on global

memory, version 3 is the parallel code on shared memory with one element per load and version 4 is the parallel code on shared memory with two elements per load.

The results were obtained for the same cases as in previously presented kernel Y, i.e. for $M = 64, 128, 512$ and $N = 2049, 4097, 8183$. The execution time of all versions was measured for 1000 iterations and on both GPUs, NVIDIA Quadro M2200 and NVIDIA Titan Xp.

On top of the bars in Figure 4.9 the speed-up of the specific version is shown. The performance of this kernel was quite better compared to the previous one. Due to the structure and launch of this kernel it appears that the speed up is affected by both the M and N parameters. However, the most dominant factor that impacts performance seems to be the number of microscopic velocities M or twice the number of threads per block for this kernel.

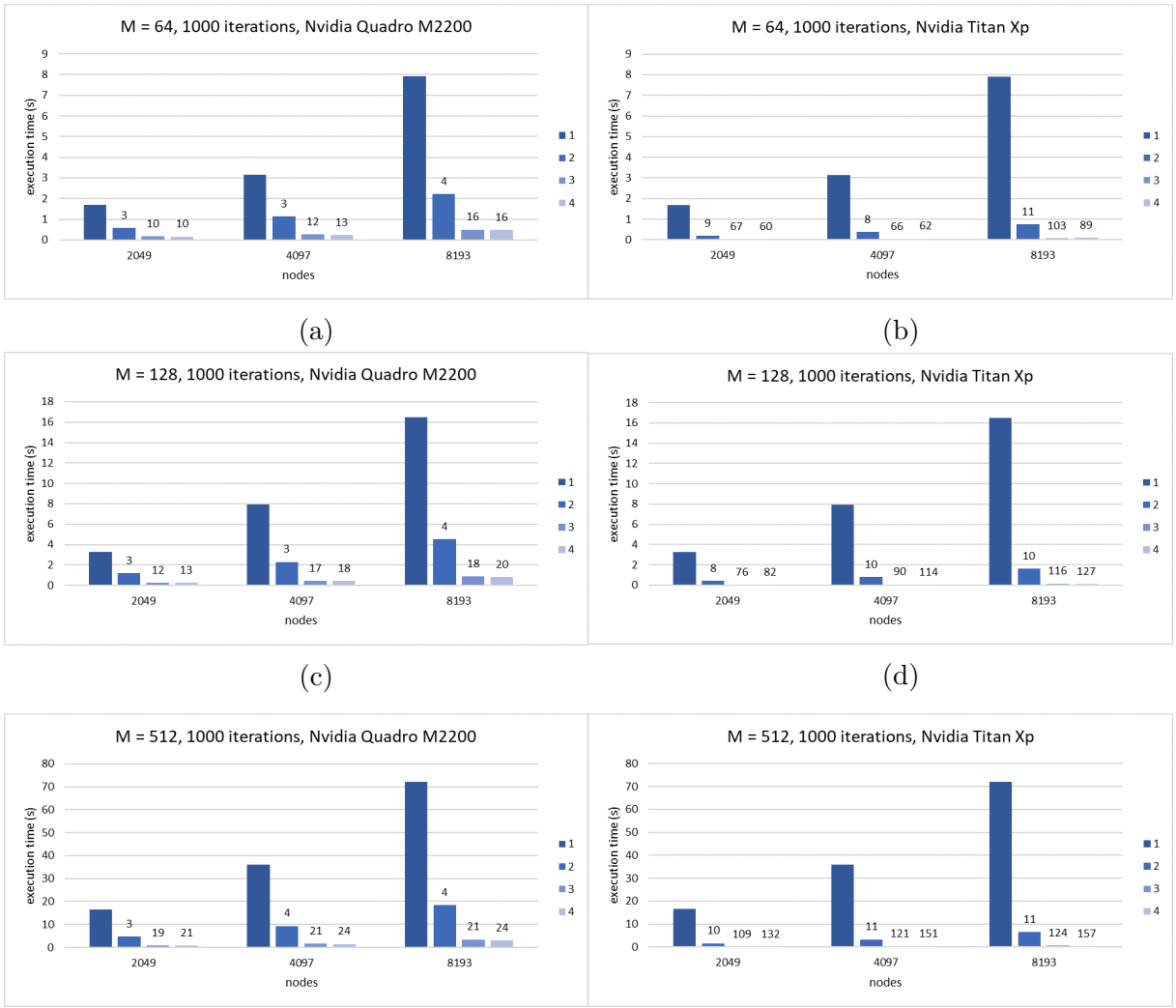


Figure 4.9: Results for kernel of macroscopic velocities.

Furthermore, it can be seen that for cases with few threads per block the efficiency of version 4 with double load per element offers little to no improvement. Nevertheless, as M is increased the speed-up seems to be affected a lot by such a small change. Observing Figure 4.9b the speed up for version 4 is lower than the one of version 3. Possible answer is that launching the kernel with 32 (version 4) instead of 64 (version 3) the achieved occupancy is too low to hide the kernel latency.

This kernel, unlike the previous one, seems to be greatly affected by the utilized GPU. The speed-up ratio of the kernel in the final case is $157/24 = 6.5$ times higher for

NVIDIA Titan Xp. A possible answer to this phenomenon is that for such a small kernel, performing few operations compared to kernel for Y, the architecture of the GPU plays an essential role in order to hide latency between data transfers. Nonetheless, this kernel is certainly memory-bound, hence memory bandwidth, which is very different between the two GPUs, has a serious impact on the execution time. In Table 4.2 results are presented for a case of 4097 spatial nodes, as similar behavior for other examined number of spatial nodes is not extremely variant.

M	**CPU**	**Nvidia Quadro M2200 speed-up**	**Nvidia Titan Xp speed-up**
64	3.142 s	13	62
128	7.95 s	18	114
512	35.992 s	24	157

Table 4.2: Results for kernel of macroscopic velocities for 4097 nodes.

Memory bandwidth and computational throughput metrics were obtained on the NVIDIA Quadro M2200 GPU, which are presented in Figures 4.10 and 4.11. Both figures have the expected curves. Memory bandwidth is asymptotically increased till a certain maximum limit is met. Moreover the increasing trend of GFLOPs reaches an upper limit as the number of threads grows. Still, although better than the previous kernel, the maximum results are far from the device's limits of 88 Gb/s and 65.6 $GFLOPs$.

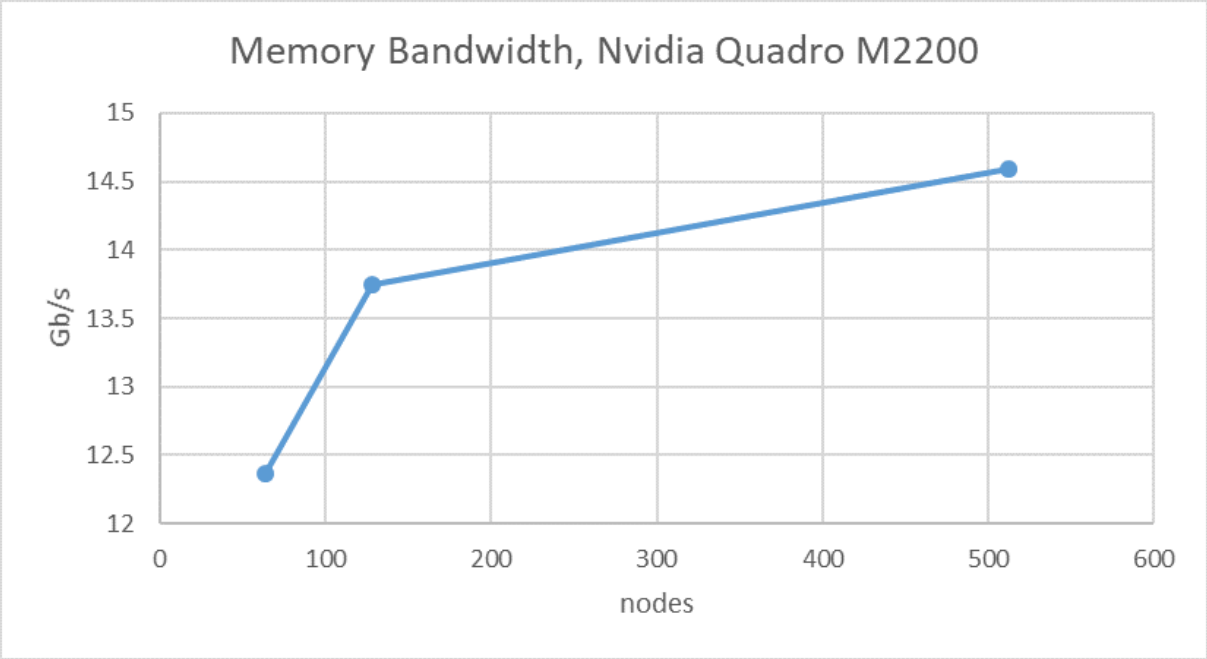


Figure 4.10: Memory bandwidth of kernel of macroscopic velocities.

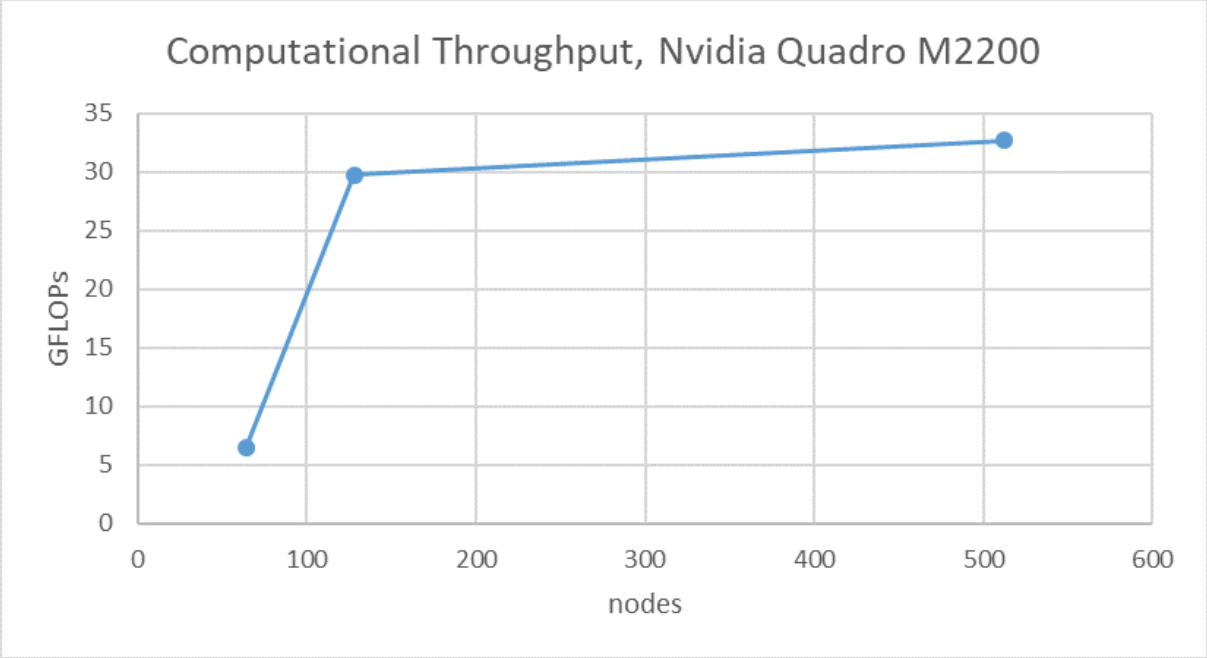
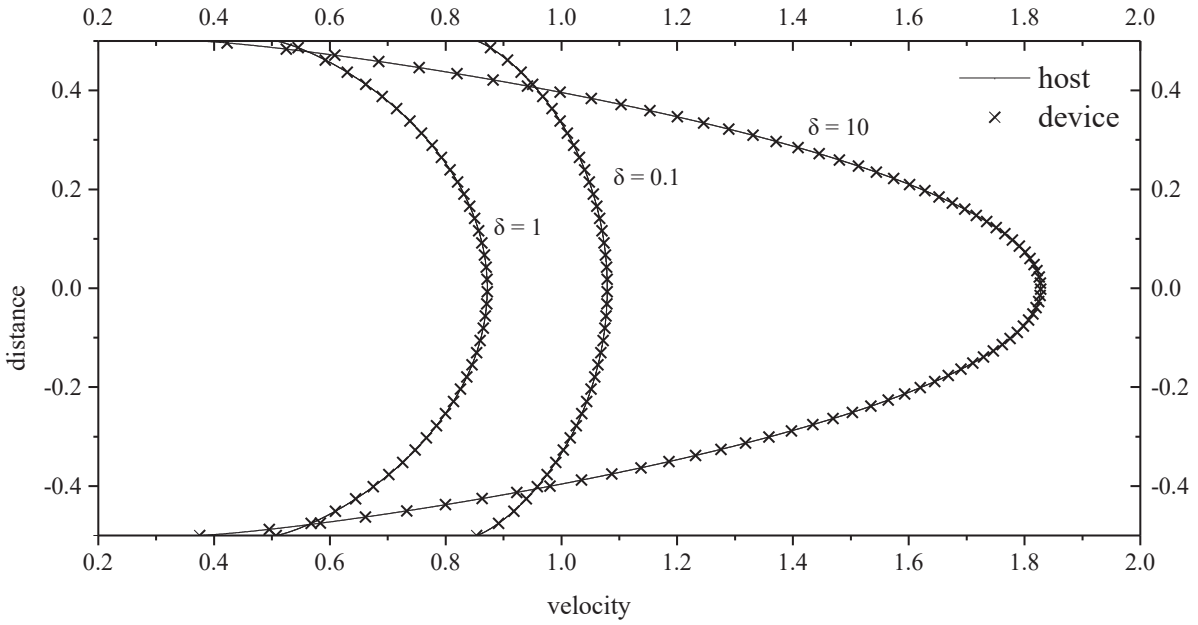


Figure 4.11: Computational throughput of kernel of macroscopic velocities.

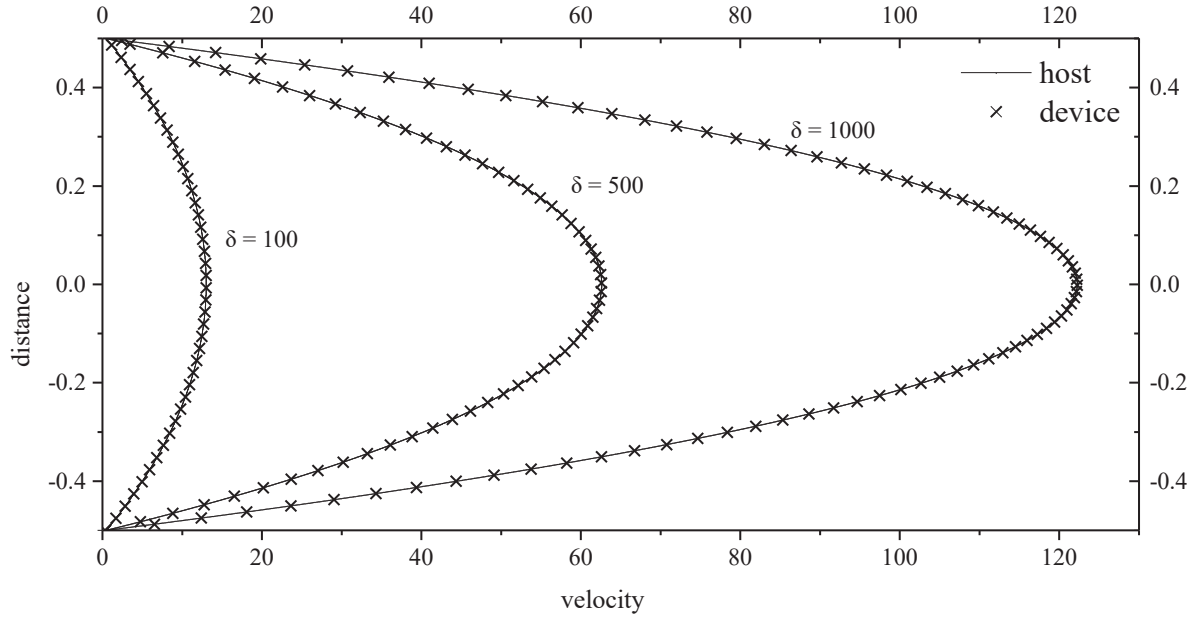
Running the PGI Visual Profiler for this kernel, it shows that no warp divergence or bank conflicts exist. However, as in the previous kernel, there is a problem when

loading two elements from global memory simultaneously. Although, as shown from the results in Figure 4.9 the version with one load per thread is far worse than two loads per thread, justifying again the conclusion that a kernel's performance may not be affected by performance throttles.

The serial code was validated from the literature, being in complete agreement for three main digits of the mass flow [45]. The complete parallel code, for the velocity profile of the flow and the mass flow rate was examined for $\delta = 0.1, 1, 10, 100, 500, 1000$. The results were in complete agreement with serial code even for double precision.



(a)



(b)

Figure 4.12: Velocity profiles for host and device versions under different values of the rarefaction parameter.

In Table 4.3 the mass flow rates are presented from the serial and parallel code, as well as data from the literature.

δ	**Serial code**	**Parallel code**	**[45]**
0.1	2.03271	2.03271	2.0314
1	1.53868	1.53868	1.5389
10	2.76863	2.76863	2.7638
100	17.6886	17.6886	-
500	83.8160	83.8160	-
1000	163.524	163.524	-

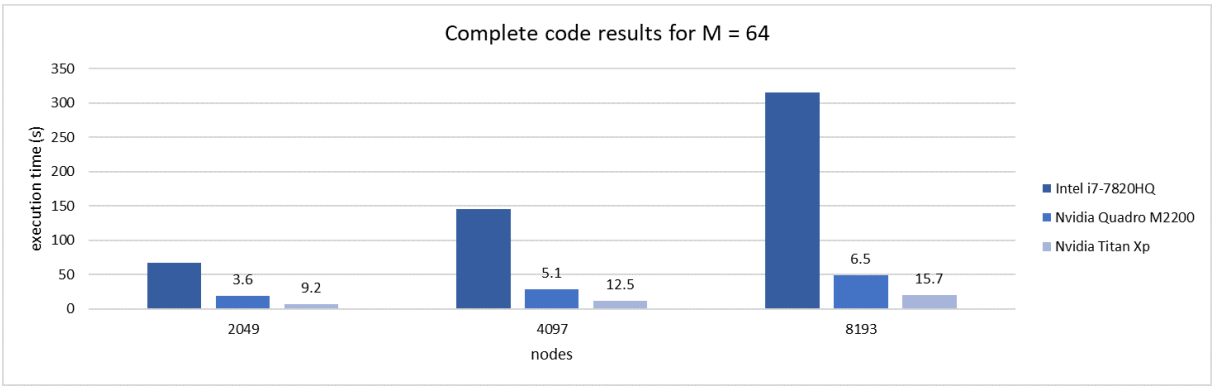
Table 4.3: Validation of serial and parallel code for mass flow rates.

The total execution time to obtain the results presented in Table 4.3 are presented in Table 4.4 on NVIDIA Quadro M2200. The measurements were obtained for a case of $M = 128$ and $N = 4097$.

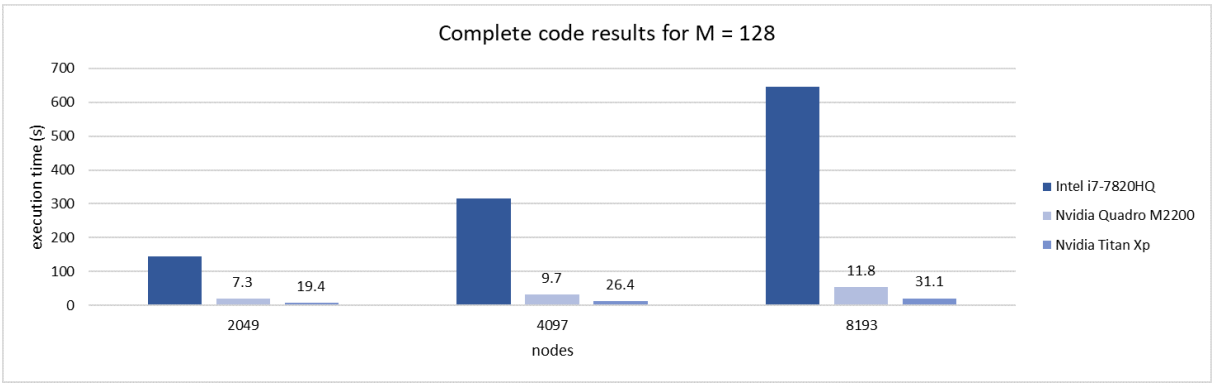
δ	**Serial code**	**Parallel code**	**Iterations**
0.1	0.176 s	32 ms	10
1	0.557 s	108 ms	32
10	7.006 s	1.213 s	394
100	321.313 s $\simeq$ 5 min	48.257 s	17941
500	4684.25 s $\simeq$ 1.3 h	645.532 s $\simeq$ 10.8 min	271065
1000	15491.104 s $\simeq$ 4.3 h	1980.474 s $\simeq$ 33 min	799360

Table 4.4: Time measurements for various deltas.

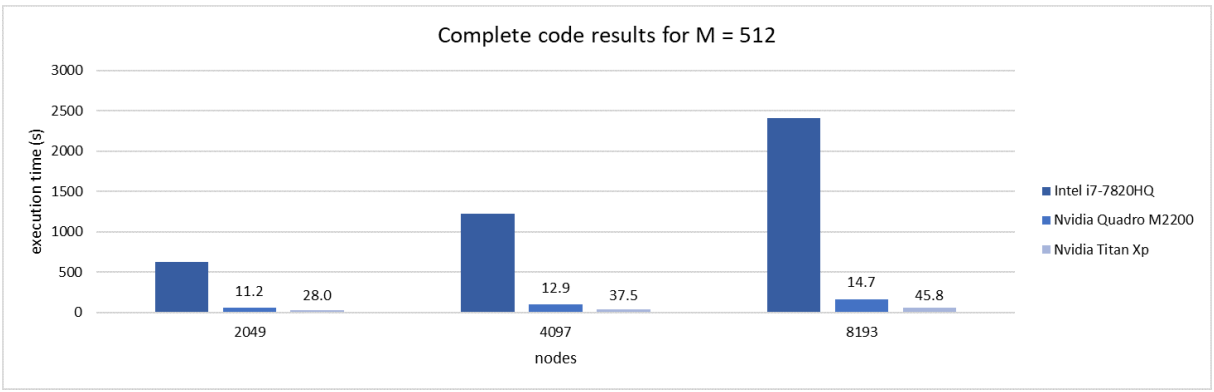
In Figure 4.13 results for the complete parallel code are presented. The measurements were performed on both GPUs and till convergence occurs for $\delta = 10$. On top of the bars the speed-up of is shown.



(a)



(b)



(c)

Figure 4.13: Results for complete parallel code.

Furthermore a similar serial and parallel code was developed on C++ to compare the two programming languages and subsequently the two compilers. The time measurements are presented on Figure 4.14 for version 2 of kernel of distribution function and version 4 for kernel of macroscopic velocities on NVIDIA Quadro M2200 GPU. The results show an

obvious superiority of CUDA Fortran, especially for kernel Y, which justifies the reason why Fortran and aged programming language is presently used and also why the CUDA developers considered it among the other modern languages. Improvement on kernel of macroscopic velocities is negligible due to the minimal operation in the main body and total execution time.

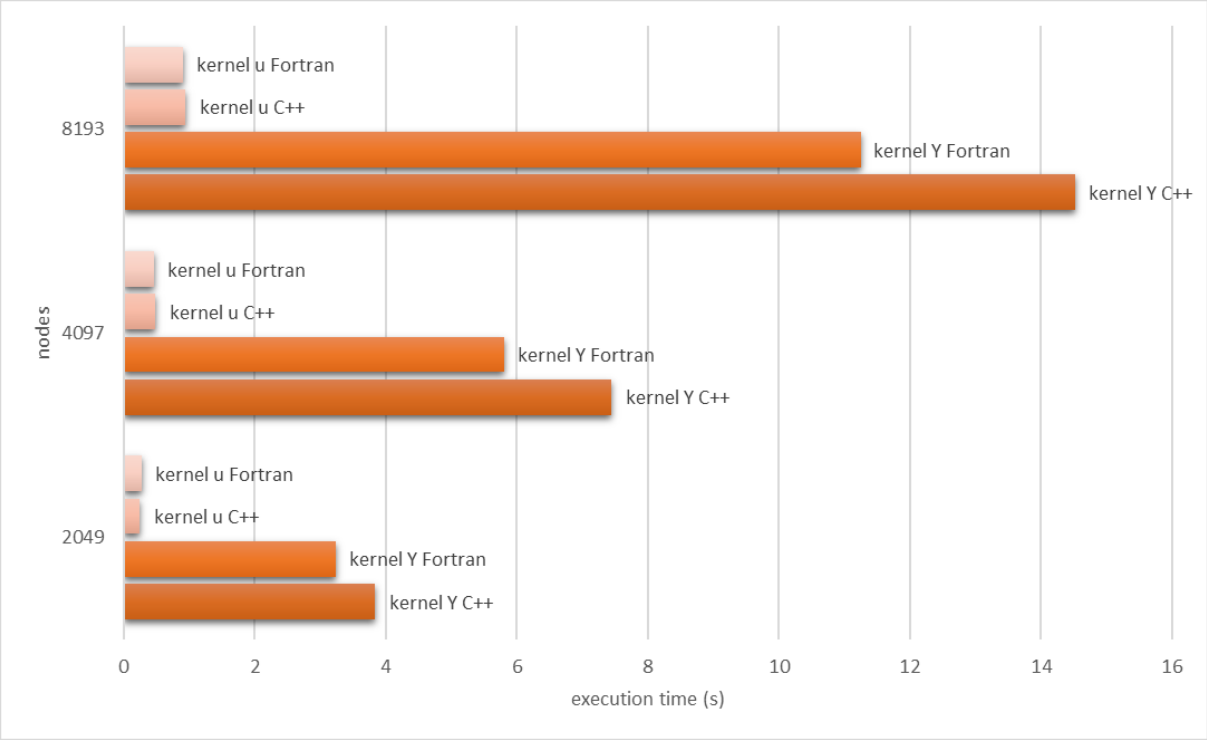


Figure 4.14: Comparison between C++ and Fortran for serial and parallel code.

Finally, the results were compared with a previous work, dealing with the same problem, showing the total improvement made from the current study [64]. The results were not greatly improved for the kernel of macroscopic velocities as it already takes minimal time, although huge improvement was observed for kernel of distribution function.

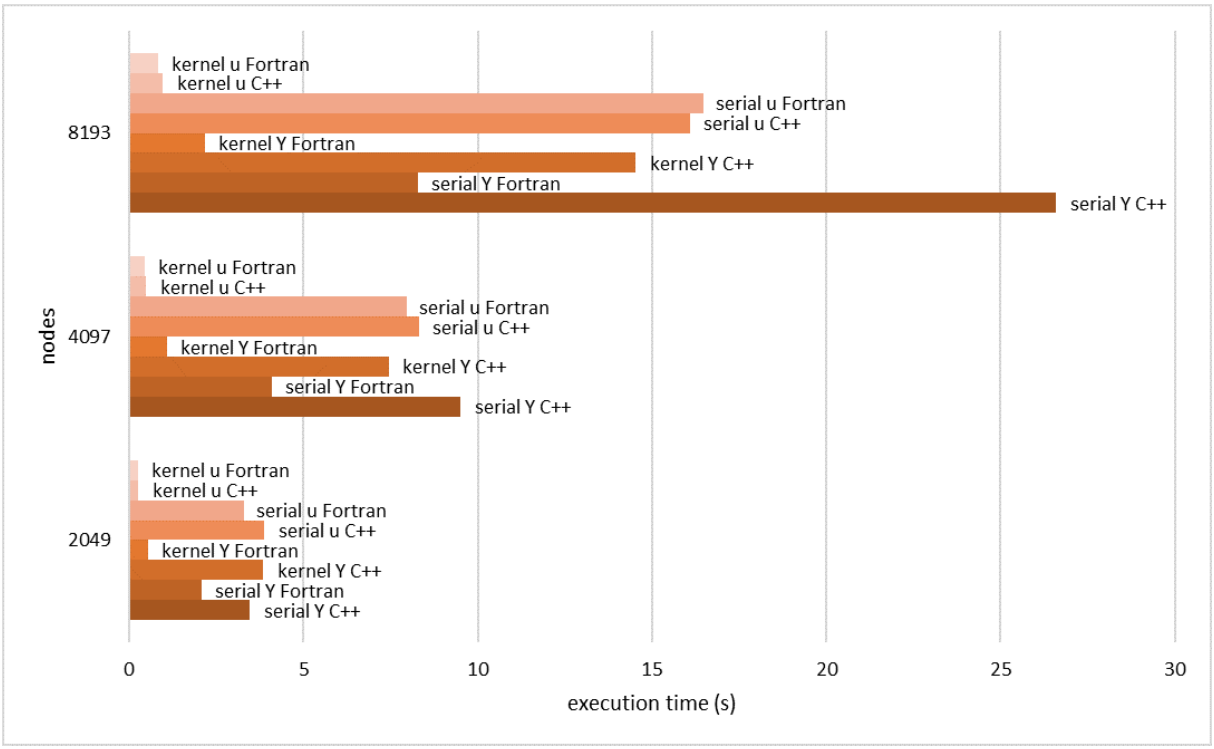


Figure 4.15: Comparison between current and previous work [64].

Chapter 5

Flow of rarefied gas in long duct

Solving a two-dimensional flow is a very computationally demanding problem, as the complexity increases significantly in comparison to the one-dimensional flow, that requires high execution times, especially for large numbers of the rarefaction parameter. Ergo, the implementation of parallel techniques to speed-up the process is not only beneficial, but also vital to obtain quick solutions. In this chapter the steady, two-dimensional flow inside a rectangular pipe of arbitrary dimensions, due to pressure gradient is studied. The solving equations are derived from the linearized BGK model. The dimensions of the duct, as well as the mesh in the x and y direction independently, is determined by the user.

5.1 Flow configuration and kinetic formulation

In Figure 5.1 a schematic representation of the problem is shown. Two vessels of the same temperature, but different pressure, are connected via a rectangular cross section pipe with length l. Due to the pressure difference on both sides of the pipe, the gas is set to motion towards the vessel with the smallest pressure. In the present work the pressure difference between the vessels is small compared to its arithmetic mean.

The governing non-dimensional BGK equation for the problem under question is Equation 5.1, where Y is the unknown distribution function and $u_z = u(x, y)$ is the macroscopic

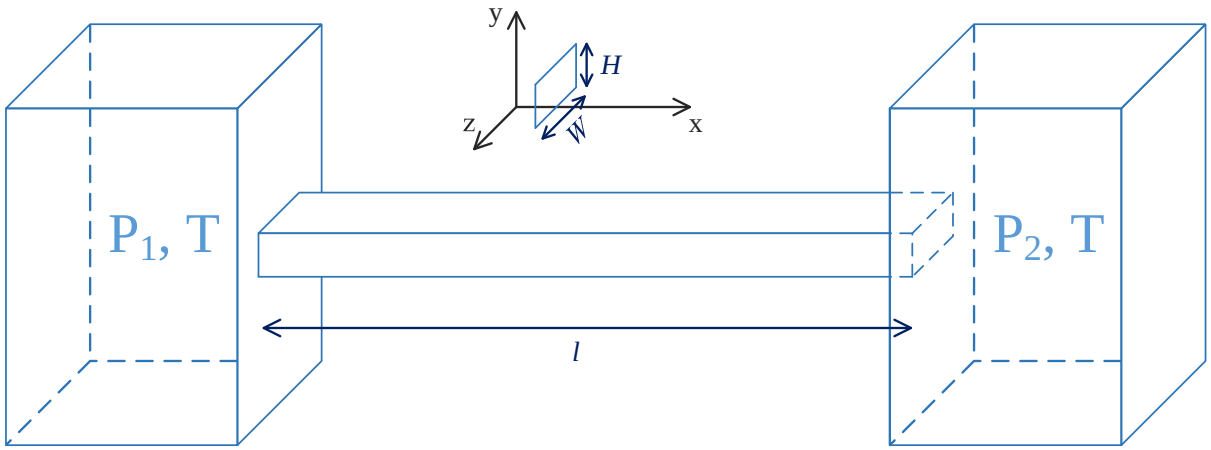


Figure 5.1: two-dimensional Poiseuille flow.

velocity. Moreover, $x \in [-W/2H, W/2H]$, $y \in [-1/2, 1/2]$ are the spatial variables in x- and y-direction, respectively and $\mu \in (-\infty, \infty)$, $\eta \in (-\infty, \infty)$ are the x- and y-component of the molecular velocity. Finally, delta is the rarefaction parameter, defined as $\delta = \dfrac{PH}{\mu u_0}$. The macroscopic velocity is defined by Equation 5.2.

$$\mu \frac{\partial Y(x,y,\mu,\eta)^{(k+1/2)}}{\partial x} + \eta \frac{\partial Y(x,y,\mu,\eta)^{(k+1/2)}}{\partial y} + \delta Y(x,y,\mu,\eta)^{(k+1/2)} = \\ = \delta u_z(x,y)^{(k)} - \frac{1}{2}, \tag{5.1}$$

$$u(x,y)^{(k+1)} = \frac{1}{\pi} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} Y(x,y,\mu,\eta)^{(k+1/2)} e^{-\mu^2-\eta^2} d\mu d\eta. \tag{5.2}$$

The boundary conditions of the problem are the following:
$Y(-\dfrac{W}{2H}, y, \mu, \eta) = 0,\ \mu > 0,\ -\infty < \eta < \infty,$
$Y(\dfrac{W}{2H}, y, \mu, \eta) = 0,\ \mu < 0,\ -\infty < \eta < \infty,$

$$Y(x,-\frac{1}{2},\mu,\eta)=0,\ -\infty<\mu<\infty,\ \eta>0,$$
$$Y(x,\frac{1}{2},\mu,\eta)=0,\ -\infty<\mu<\infty,\ \eta<0.$$

Given the solution of macroscopic velocities, the non-dimensional mass flow rate can be computed as follows:

$$G=2\frac{H}{W}\int_{-1/2}^{1/2}\int_{-W/2H}^{W/2H}u(x,y)dxdy. \tag{5.3}$$

By means of computational efficiency is is advantageous to convert the molecular velocities space into a Polar coordinate system, which is shown in Figure 5.2. Ergo, one ca define the variables $\zeta=\sqrt{\mu^2+\eta^2}$, which is the magnitude of the molecular velocity's vector and $\theta=ArcTan\left(\frac{\mu}{\eta}\right)$, which is the angle of the molecular velocity's vector.

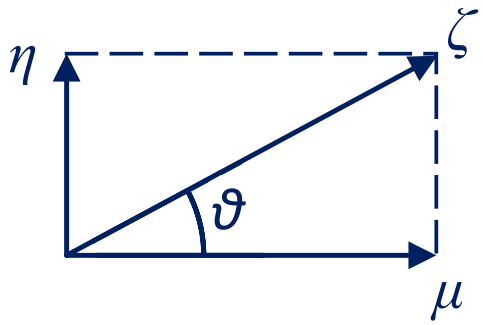


Figure 5.2: Molecular velocities space on a polar coordinate system.

Thus, one obtains the aforementioned variables μ, η as $\mu=\zeta cos\theta$, $\eta=\zeta sin\theta$.

Due to this transformation, Equations 5.1 and 5.2 are converted to Equations 5.4 and 5.5.

$$\zeta cos\theta\frac{\partial Y(x,y,\zeta,\theta)^{(k+1/2)}}{\partial x}+\zeta sin\theta\frac{\partial Y(x,y,\zeta,\theta)^{(k+1/2)}}{\partial y}+\delta Y(x,y,\zeta,\theta)^{(k+1/2)}= \tag{5.4}$$
$$=\delta u_z(x,y)^{(k)}-\frac{1}{2},$$

$$u(x,y)^{(k+1)} = \frac{1}{\pi}\int_0^{2\pi}\int_0^{\infty} Y(x,y,\zeta,\theta)^{(k+1/2)} e^{-\zeta^2}\zeta d\zeta d\theta. \tag{5.5}$$

5.2 Serial code

To obtain the solution, an iterative procedure takes place, till convergence. The iterations terminate either when the relative error drops below a user-defined maximum tolerance, or when the iterations surpass a maximum limit.

The iterations begin by assuming the macroscopic velocities $u(x,y)$ at the right hand side of Equation 5.4 and solving for the distribution function $Y(x,y,\zeta,\theta)$. Afterwards the solution of $Y(x,y,\zeta,\theta)$ is substituted to Equation 5.5 and the new value of $u(x,y)$ is found. The new solution of $u(x,y)$ is being compared with the one from the previous iteration, in order to check for convergence.

A set of discrete molecular velocities p_m, with $m = 1,2,...,M$, is defined at the beginning of the program. In this case the arithmetic scheme that was chosen is the Gauss-Legendre and thus each root of the Legendre polynomials represents one molecular wight. The number of roots and the corresponding weights is determined as an input from the user. Half of the pairs –nodes and weights– need to be administered as the opposites are the same values but negative, hence this is done independently inside the program.

```
1: open(1, file ='g128.dat', status ="old")
2: do i = 1, M / 2
3:     read(1, *) CC(i), WW(i)      ! Positive velocities
4:     CC(i + M / 2) = - CC(i)      ! Negative velocities
5:     WW(i + M / 2) = WW(i)        ! Weight factors
6: end do
```

Listing 5.2.1: Reading Gauss-Legendre roots and weights.

The file name corresponds to the total number of Gauss-Legendre roots (both positive and negative) that will be read and eventually used during execution. Only half values

—roots and weights— will be used, thus despite the file name's description, only half values should be found inside this ".dat" file and the negatives are created inside the *d–loop*. Altering this file name various number of Gauss-Legendre roots and weights can be inputted in the code.

As in chapter 3 a transformation must occur so that the discreet molecular velocities will belong to the field $[0,\infty)$ rather than $[-1,1]$. This can easily be accomplished by introducing a new variable ζ as in Equation 5.6. It is important to perform this step as later on it will be proved that the derived equations are of the exact same structure. As a result only the discrete set of $c \in [0,\infty)$ should be used.

$$\zeta = \frac{1+p}{1-p}, \qquad d\zeta = \frac{2dp}{(1-p)^2}. \tag{5.6}$$

The new variable ζ introduces a a set of discreet velocities ζ_m, $m = 1,2,...,M$, in $[0,\infty)$.

```
1: do i = 1, M
2:     C(i) = (1.0_fpKind + CC(i)) / (1.0_fpKind - CC(i))
3:     W(i) = 2.0_fpKind * WW(i) / (1.0_fpKind - CC(i)) ** 2.0_fpKind
4: end do
```

Listing 5.2.2: Transformation of discrete velocities and weighting factors for the integration.

Moreover, a set of discrete angles θ_n, $n = 1,2,...,N \in [0,2\pi]$ is defined, in such manner as no angle to be equal to $\kappa\frac{\pi}{2}$, $\kappa = 0,1,2,3,4$.

```
1: Wt=2.0_fpKind*Pi/Nt
2:
3: Do L=1,Nt
4:     Theta(L)=Wt*(L-0.5_fpKind)
5: end do
```

Listing 5.2.3: Discrete angles

As for the spatial mesh, the domain $x \in [-W/2H, W/2H]$, $y \in [-1/2, 1/2]$ is divided into I equal intervals of length $h_x = W/(H \cdot I)$ in the x-direction and J equal intervals of length $h_y = 1/J$ in the y-direction. Also, the mid point of each interval is defined, as shown in Figure 5.3.

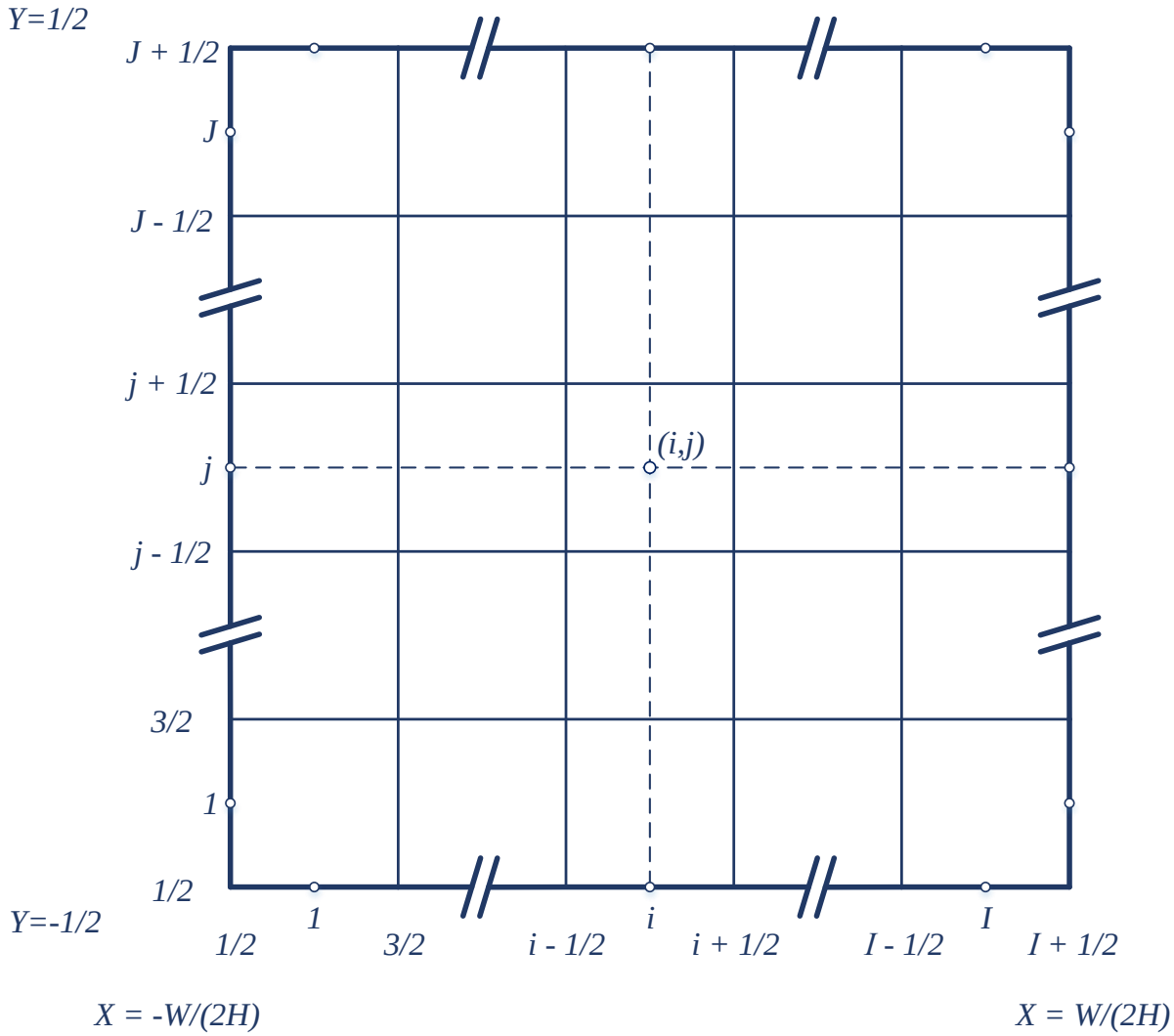


Figure 5.3: Spatial grid of two-dimensional Poiseuille flow.

Setting $Y(x,y,\zeta,\theta)|_{i,j,m,n} = Y(x_i, y_j, \zeta_m, \theta_n) = Y_{i,j,m,n}$, the kinetic equation can be discretized at the node (i,j,m,n), $i = 1,2,...,I$, $j = 1,2,...,J$, $m = 1,2,...,M$, $n = 1,2,...,N$,

as in Equation 5.7.

$$\zeta_m cos\theta_n \frac{\partial Y(x,y,\zeta,\theta)^{(k+1/2)}}{\partial x}\bigg|_{i,j,m,n} + \zeta_m sin\theta_n \frac{\partial Y(x,y,\zeta,\theta)^{(k+1/2)}}{\partial y}\bigg|_{i,j,m,n} + \\ + \delta Y(x,y,\zeta,\theta)^{(k+1/2)}\bigg|_{i,j,m,n} = \delta u_z(x,y)^{(k)}\bigg|_{i,j,m,n} - \frac{1}{2}. \tag{5.7}$$

By making the following approximations, Equations 5.8, 5.9, 5.10, 5.11, using Taylor series, Equation 5.12 is deduced.

$$\frac{\partial Y}{\partial x}\bigg|_{i,j,m,n} = \frac{1}{2h_x}\left[Y_{i+\frac{1}{2},j+\frac{1}{2},m,n} + Y_{i+\frac{1}{2},j-\frac{1}{2},m,n} - Y_{i-\frac{1}{2},j+\frac{1}{2},m,n} - Y_{i-\frac{1}{2},j-\frac{1}{2},m,n}\right] + O\left[h^2\right], \tag{5.8}$$

$$\frac{\partial Y}{\partial y}\bigg|_{i,j,m,n} = \frac{1}{2h_y}\left[Y_{i+\frac{1}{2},j+\frac{1}{2},m,n} + Y_{i-\frac{1}{2},j+\frac{1}{2},m,n} - Y_{i+\frac{1}{2},j-\frac{1}{2},m,n} - Y_{i-\frac{1}{2},j-\frac{1}{2},m,n}\right] + O\left[h^2\right], \tag{5.9}$$

$$Y_{i,j,m,n} = \frac{1}{4}\left[Y_{i+\frac{1}{2},j+\frac{1}{2},m,n} + Y_{i-\frac{1}{2},j+\frac{1}{2},m,n} + Y_{i+\frac{1}{2},j-\frac{1}{2},m,n} + Y_{i-\frac{1}{2},j-\frac{1}{2},m,n}\right] + O\left[h^2\right], \tag{5.10}$$

$$u_{i,j} = \frac{1}{4}\left[u_{i+\frac{1}{2},j+\frac{1}{2}} + u_{i-\frac{1}{2},j+\frac{1}{2}} + u_{i+\frac{1}{2},j-\frac{1}{2}} + u_{i-\frac{1}{2},j-\frac{1}{2}}\right] + O\left[h^2\right], \tag{5.11}$$

$$\begin{aligned}
&\frac{\zeta_m cos\theta_n}{2h_x}\left[Y^{(k+1/2)}_{i+\frac{1}{2},j+\frac{1}{2},m,n}+Y^{(k+1/2)}_{i+\frac{1}{2},j-\frac{1}{2},m,n}-Y^{(k+1/2)}_{i-\frac{1}{2},j+\frac{1}{2},m,n}-Y^{(k+1/2)}_{i-\frac{1}{2},j-\frac{1}{2},m,n}\right]+\\
&+\frac{\zeta_m sin\theta_n}{2h_y}\left[Y^{(k+1/2)}_{i+\frac{1}{2},j+\frac{1}{2},m,n}+Y^{(k+1/2)}_{i-\frac{1}{2},j+\frac{1}{2},m,n}-Y^{(k+1/2)}_{i+\frac{1}{2},j-\frac{1}{2},m,n}-Y^{(k+1/2)}_{i-\frac{1}{2},j-\frac{1}{2},m,n}\right]+\\
&+\frac{\delta}{4}\left[Y^{(k+1/2)}_{i+\frac{1}{2},j+\frac{1}{2},m,n}+Y^{(k+1/2)}_{i-\frac{1}{2},j+\frac{1}{2},m,n}+Y^{(k+1/2)}_{i+\frac{1}{2},j-\frac{1}{2},m,n}+Y^{(k+1/2)}_{i-\frac{1}{2},j-\frac{1}{2},m,n}\right]=\\
&=\frac{\delta}{4}\left[u^{(k)}_{i+\frac{1}{2},j+\frac{1}{2}}+u^{(k)}_{i-\frac{1}{2},j+\frac{1}{2}}+u^{(k)}_{i+\frac{1}{2},j-\frac{1}{2}}+u^{(k)}_{i-\frac{1}{2},j-\frac{1}{2}}\right]-\frac{1}{2}.
\end{aligned} \tag{5.12}$$

The above system is solved by following the particle trajectories.

The following variables are introduced, to make the system of equations more readable. Distribution function's coefficients are $Td00$, $Td01$, $Td10$ and $Td11$ and the right hand side's of Equation 5.12 is CoF.

$$Td00 = -\frac{\zeta_m cos\theta_n}{2h_x}-\frac{\zeta_m sin\theta_n}{2h_y}+\frac{\delta}{4}, \tag{5.13}$$

$$Td01 = -\frac{\zeta_m cos\theta_n}{2h_x}+\frac{\zeta_m sin\theta_n}{2h_y}+\frac{\delta}{4}, \tag{5.14}$$

$$Td10 = \frac{\zeta_m cos\theta_n}{2h_x}-\frac{\zeta_m sin\theta_n}{2h_y}+\frac{\delta}{4}, \tag{5.15}$$

$$Td11 = \frac{\zeta_m cos\theta_n}{2h_x}+\frac{\zeta_m sin\theta_n}{2h_y}+\frac{\delta}{4}, \tag{5.16}$$

$$CoF = \frac{\delta}{4}\left[u^{(k)}_{i+\frac{1}{2},j+\frac{1}{2}}+u^{(k)}_{i-\frac{1}{2},j+\frac{1}{2}}+u^{(k)}_{i+\frac{1}{2},j-\frac{1}{2}}+u^{(k)}_{i-\frac{1}{2},j-\frac{1}{2}}\right]^{(k)}-\frac{1}{2}. \tag{5.17}$$

Equation 5.12 can now be divided into 4 subsequent equations according to the quarter the angle is located. The new value of the distribution function is defined accordingly.

- For $0 < \theta_n < \dfrac{\pi}{2}$

$$Y^{(k+1/2)}_{i+\frac{1}{2},j+\frac{1}{2},m,n} =$$

$$[Td11]^{-1}\left\{-Td00\cdot Y^{(k+1/2)}_{i-\frac{1}{2},j-\frac{1}{2},m,n} - Td10\cdot Y^{(k+1/2)}_{i+\frac{1}{2},j-\frac{1}{2},m,n} \right. \tag{5.18}$$

$$\left. -Td01\cdot Y^{(k+1/2)}_{i-\frac{1}{2},j+\frac{1}{2},m,n} + CoF\right\},$$

$i = 1,2,...,I,\ j = 1,2,...,J,\ m = 1,2,...,M,\ n = 1,2,...,N/4$ with boundary conditions $Y_{\frac{1}{2},j,m,n} = 0$ and $Y_{i,\frac{1}{2},m,n} = 0$

- For $\dfrac{\pi}{2} < \theta_n < \pi$

$$Y^{(k+1/2)}_{i-\frac{1}{2},j+\frac{1}{2},m,n} =$$

$$[Td01]^{-1}\left\{-Td00\cdot Y^{(k+1/2)}_{i-\frac{1}{2},j-\frac{1}{2},m,n} - Td10\cdot Y^{(k+1/2)}_{i+\frac{1}{2},j-\frac{1}{2},m,n} \right. \tag{5.19}$$

$$\left. -Td11\cdot Y^{(k+1/2)}_{i+\frac{1}{2},j+\frac{1}{2},m,n} + CoF\right\},$$

$i = I, I-1,...,1,\ j = 1,2,...,J,\ m = 1,2,...,M,\ n = N/4,...,N/2$ with boundary conditions $Y_{I+\frac{1}{2},j,m,n} = 0$ and $Y_{i,\frac{1}{2},m,n} = 0$

- For $\pi < \theta_n < \dfrac{3\pi}{2}$

$$Y^{(k+1/2)}_{i-\frac{1}{2},j-\frac{1}{2},m,n} =$$

$$[Td00]^{-1}\Big\{-Td10\cdot Y^{(k+1/2)}_{i+\frac{1}{2},j-\frac{1}{2},m,n} - Td01\cdot Y^{(k+1/2)}_{i-\frac{1}{2},j+\frac{1}{2},m,n} \tag{5.20}$$

$$-Td11\cdot Y^{(k+1/2)}_{i+\frac{1}{2},j+\frac{1}{2},m,n} + CoF\Big\},$$

$i = 1,2,...,I$, $j = J, J-1,...,1$, $m = 1,2,...,M$, $n = N/2,...,3N/4$ with boundary conditions $Y_{I+\frac{1}{2},j,m,n} = 0$ and $Y_{i,J+\frac{1}{2},m,n} = 0$

- For $\dfrac{3\pi}{2} < \theta_n < 2\pi$

$$Y^{(k+1/2)}_{i+\frac{1}{2},j-\frac{1}{2},m,n} =$$

$$[Td10]^{-1}\Big\{-Td00\cdot Y^{(k+1/2)}_{i-\frac{1}{2},j-\frac{1}{2},m,n} - Td01\cdot Y^{(k+1/2)}_{i-\frac{1}{2},j+\frac{1}{2},m,n} \tag{5.21}$$

$$-Td11\cdot Y^{(k+1/2)}_{i+\frac{1}{2},j+\frac{1}{2},m,n} + CoF\Big\},$$

$i = 1,2,...,I$, $j = J, J-1,...,1$, $m = 1,2,...,M$, $n = 3N/4,...,N$ with boundary conditions $Y_{\frac{1}{2},j,m,n} = 0$ and $Y_{i,J+\frac{1}{2},m,n} = 0$

When the solution of Y is obtained, an integration over μ and η outputs u. For the magnitude ζ the Gauss-Legendre Integration is used, whereas for the angle θ the trapezoidal rule.

$$u(x,y)^{(k+1)} = \frac{1}{\pi}\int_0^{2\pi}\int_0^{\infty} Y(x,y,\zeta,\theta)^{(k+1/2)} e^{-\zeta^2}\zeta d\zeta d\theta, \tag{5.22}$$

or in discrete form:

$$u_i^{(k+1)} = \frac{1}{\pi} \sum_{n=1}^{N} \sum_{m=1}^{M} Y_{i,j,\zeta_m,\theta_n}^{(k+1/2)} e^{-\zeta_m^2} \zeta_m w_m w_n. \tag{5.23}$$

```
do i=1,Nx
    do j=1,Ny
        s=0
        do k=1,Nc
            do l=1,Nt
                s=s+Y(k,l,i,j)*C(k)*W(k)*Wt*EXP(-C(k)**2)
            end do
        end do
        u(i,j)=s/Pi
    end do
end do
```

Listing 5.2.4: Calculation of macroscopic velocities.

Afterwards, by integrating the macroscopic velocity over the spatial domain the mass flow rate is obtained.

$$\begin{aligned} G =& 2\frac{H}{W} \int_{-1/2}^{1/2} \int_{-W/2H}^{W/2H} u(x,y)dxdy = 2h_x h_y \frac{H}{W} \left[\frac{1}{4} (u_{11} + u_{1J} + u_{I1} + u_{IJ}) + \right. \\ &\left. + \frac{1}{2} \left(\sum_{i=2}^{I-1} \left(u_{i1} + u_{iJ} \right) + \sum_{j=2}^{J-1} \left(u_{1j} + u_{Ij} \right) \right) + \sum_{i=2}^{I-1} \sum_{j=2}^{J-1} u_{ij} \right]. \end{aligned} \tag{5.24}$$

```
do i = 2, Nx - 1
    do j = 1, Ny - 2
```

```
3:            W_i(j * Nx + i) = 1.0_fpKind
4:        end do
5:    end do
6:
7:    do j = 1, Ny - 2
8:        W_i(j * Nx + 1) = 0.5_fpKind
9:        W_i((j + 1) * Nx) = 0.5_fpKind
10:   end do
11:
12:   do i = 2, Nx - 1
13:       W_i(i) = 0.5_fpKind
14:       W_i((Ny - 1) * Nx + i) = 0.5_fpKind
15:   end do
16:
17:   W_i(1) =0.25_fpKind
18:   W_i(Nx) =0.25_fpKind
19:   W_i((Ny - 1) * Nx + 1) = 0.25_fpKind
20:   W_i(Nx *Ny) = 0.25_fpKind
21:
22:   S = 0.0_fpKind
23:   do i = 1, Nx * Ny
24:       S = S + W_i(i) * u(i)
25:   end do
26:
27:   Flow_Rate =- 2.0_fpKind * Dx * Dy * ratio * S
```

Listing 5.2.5: Calculation of mass flow rate.

5.3 Implementation on GPU

In favor of easier management of the stored arrays on GPU, the array of distribution function is translated from four-dimensional `Y(Nc,Nt,Nx,Ny)` to a three-dimensional `Y(Nc*Nt,Nx,Ny)`, with the angles and magnitudes fused into one dimension. The pattern of the stored data is that for every angle, all magnitudes are sequentially stored. In a similar way the array of macroscopic velocities is transformed from two-dimensional `u(Nx,Ny)` to one-dimensional `u(Nx*Ny)` with a pattern that for every node in y-direction, all Nx data of x-direction are unfolded. At the beginning, before the iterations are initiated, all necessary arrays are being copied to GPU's global memory. Even for cases of few nodes, array `Y` can become several Gb in size, thus carefulness is required during device allocation. Constant memory is used for differentials in the x and y direction (parameters `Dx_c`, `Dy_c`), the rarefaction parameter `delta_c`, and π (parameter `Pi_c`), as well as the term $Wt = 2\pi/Nt$ (parameter `Wt_c`).

To obtain the solution on GPU, four kernels have been developed. The first one calculates the distribution function, the second and third contribute to the calculation of macroscopic velocities and a forth kernel helps to obtain the total residual on device, making unnecessary the transfer of array `u` from device to host.

5.3.1 Kernel of Distribution Function

The parallelization strategy on GPU for the kernel of distribution function is that each thread solves for one magnitude (`Nc` in total) and each block is assigned to one angle (`Nt` in total). In Figure 5.4 this idea is shown schematically. The appropriate number of magnitudes for this problem will never exceed the 512 in total, whereas 128 or 256 are more than enough, thus they will never exceed the upper limit of threads per block. In order to further increase the parallelism, blocks are divided into four groups, which subsequently are assigned to solve for each one of the four polar coordinate system quarters simultaneously. Therefore, $1 \leq$ `blockIdx%x` $\leq Nt/4$ blocks are assigned to the first quarter, $Nt/4 \leq$ `blockIdx%x` $\leq Nt/2$ to the second quarter, $Nt/2 \leq$ `blockIdx%x` $\leq 3Nt/4$ to the

third quarter and $3Nt/4 \leq$ `blockIdx%x` $\leq Nt$ to the fourth and final quarter. Inside the kernel, the separation of each quarter happens with an *if-else* statement.

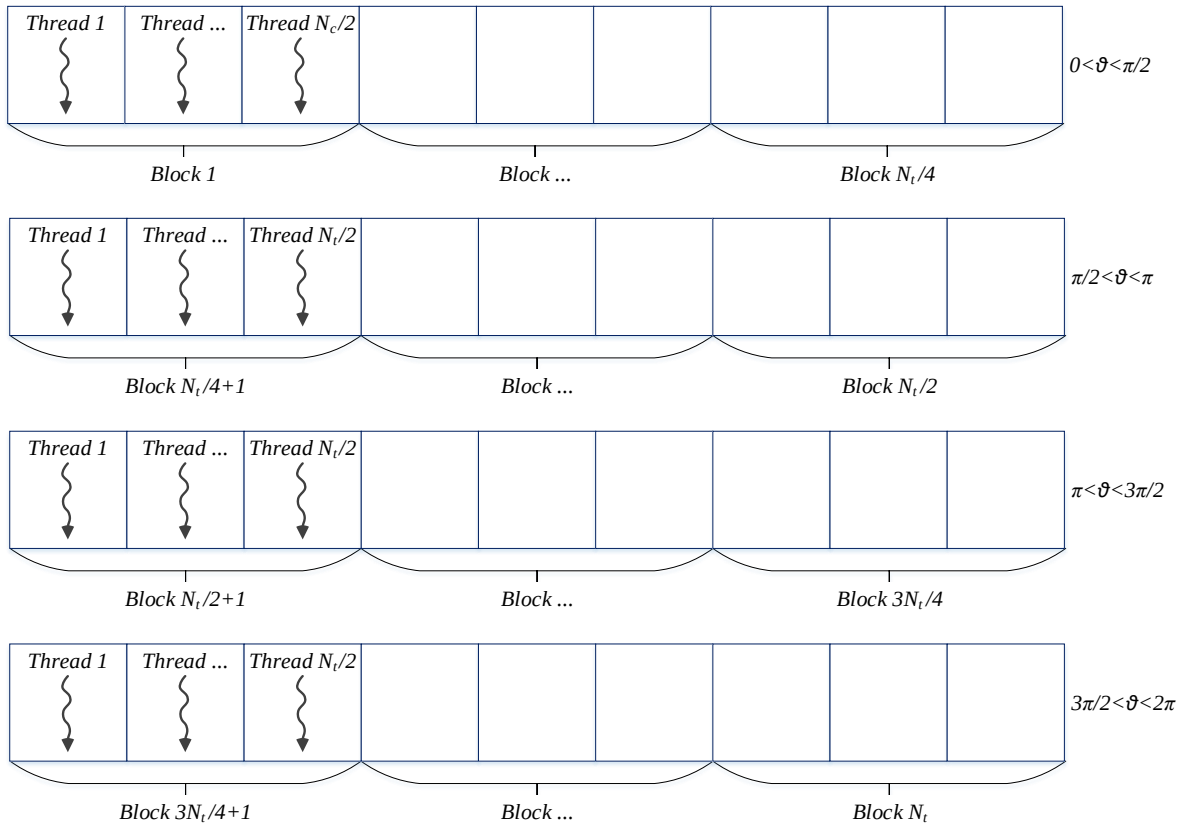


Figure 5.4: Schematic representation of distribution function kernel parallelization strategy.

The kernel has access to arrays `Y` of distribution function, `u` of macroscopic velocities and `C` of discrete magnitudes. At the beginning, before the *if-statement*, the variables $Td00$, $Td10$, $Td01$, $Td11$ are calculated and are placed to registers. For each magnitude and angle two nested *do-loops* the distribution function is solved for the spatial grid, according to Equations 5.18, 5.19, 5.20, 5.21. The outside *do-loop* is designated to y-direction and the inside to x-direction.

Examining the kernel with PGI Visual Profiler shows that its performance may be limited by register usage and is limited to run concurrently warps of approximately 6 blocks for 128 threads per block and 12 blocks for 64 threads per blocks. Compiling the program with the `-Mcuda=ptxinfo`, shows that 86 registers are being used, whereas 36 bytes are

spilled to local memory due to dynamically accessed arrays. Register usage may be a limited factor, but it is inevitable when a kernel absolutely requires that much registers. For this kernel, placing variables on shared memory, or reducing register usage during compiling with the flag `-Mcuda=maxregcount:<n>` –where n is the preferred number of registers–, kernel performance worsens. Therefore, the message from PGI Visual Profiler warns the programmer to be careful during development, not to bind many registers, but reducing the required by the problem registers during compiling will not guarantee faster execution.

To further speed-up the calculation of distribution function shared memory is used. Observing the calculation of `Y`, four values of the macroscopic velocities must be summed. In the two-dimensional array of macroscopic velocities these four values are bordering. Hence, two, one-dimensional arrays are allocated to shared memory, `usA`, `usB`. In each of the two arrays is assigned the sum of two consecutive elements of one column of the two-dimensional array `u`. The size of `usA`, `usB` is $Nx-1$. As the index $j=1,...,Ny$ changes the next, consecutive column of array `u` is required. Therefore, data from `usB` are being transferred to `usA` and the new data are copied to `usB` from global memory. Since shared memory is a very fast on-chip memory, transferring previously needed data from shared to shared memory banks is extremely fast. As a result only one access to global memory occurs per iteration on y direction. In Figure 5.5 the idea of shared memory usage is schematically shown.

The utilization of shared memory enhances the speed-up of the kernel, especially when the number of spatial nodes increases in x coordinate. Since Fortran saves arrays linearly, per column, in the initial version with only global memory being used, uncoalesced accesses occur when reading values of `u` from the two columns. For example, for the first quarter the initial values of `u` that must be read are located into the array by the indices 1, 2, N_x+1 and N_x+2, thus the thread must access data from global memory banks that are far away from each other. On the contrary, shared memory does not have this problem, as explained in section 2.5.3. Therefore, the utilization of shared memory does not only decrease the run time due to being considerably faster in memory transactions,

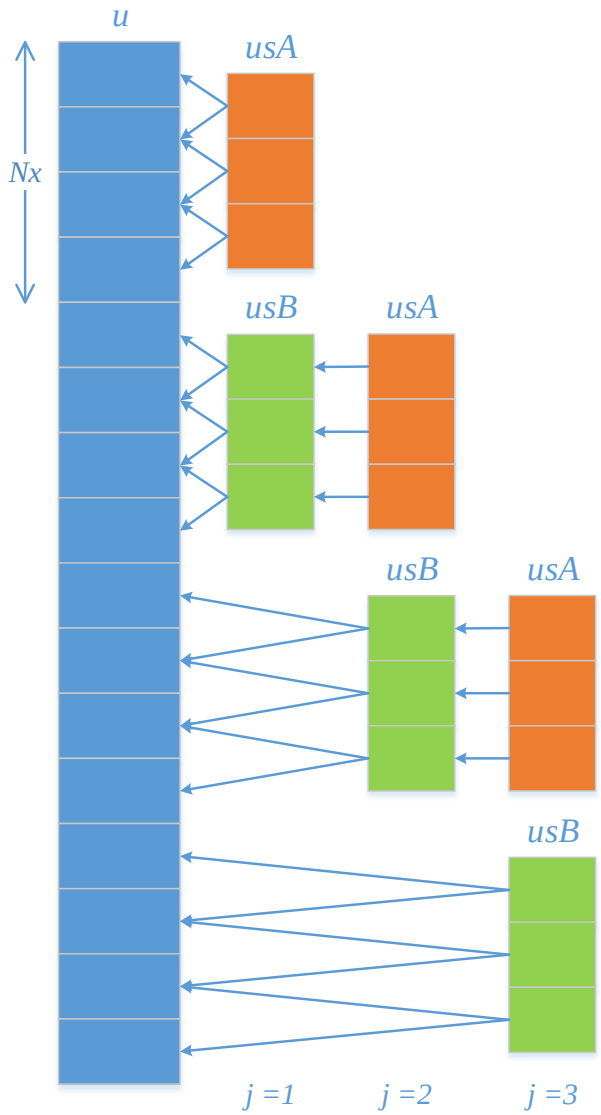


Figure 5.5: Schematic representation of shared memory utilization for the calculation of distribution function.

but also because it corrects the way memory accesses take place in global memory.

It is important to state that copying more columns or even the whole array (for cases with coarse spatial grid) to shared memory will result in a degraded performance, since less blocks can be concurrently executed, due to limited resources. One should have in mind that using all 48 kB of shared memory, an SM is equipped with, for a single block will result to only the warps of one block being concurrently executed at a time on a single SM. For this specific problem, since the size of a shared memory array is $N_x - 1$, in

case 257 or 513 cells are used in x-direction, the total capacity of shared memory being allocated is $2 \times 257 \times 8 = 4112$ Bytes or $2 \times 513 \times 8 = 8208$ Bytes, where the factor 2 refers to the number of arrays and 8 to double precision. As explained previously, the kernel for 64 threads per block runs approximately 12 blocks per SM due to other limitations. Hence, for the case mentioned, of $N_x = 257$ cells, the number of blocks per SM being simultaneously executed does not reduce, as 4112×12 results in 48kB used, which is the total capacity of shared memory, whereas for $N_x = 513$ cells the number of blocks per SM is already reduced to half. The latter result would also occur for the initial case of $N_x = 257$, if not two but four columns were loaded in shared memory. As a result, a slight increase in the number of, $N_x - 1$ in size, shared memory arrays significantly reduces the blocks that can be executed simultaneously per SM, resulting in an important performance degradation. Concluding, in order to fully exploit the benefits of shared memory but not limit significantly the number of active warps per SM, the version of loading two columns at a time prevailed.

Examining the kernel of distribution function, that uses shared memory, with PGI Visual Profiler outputs a probable coalescing problem. When reading two successive elements from the array of macroscopic velocities and placing them to shared memory by one thread, uncoalesced accesses occur. As described at chapter 1, global memory can be accessed via 32-, 64-, or 128-byte transactions and best performance is achieved when threads in a warp access data in as few memory transactions as possible. In this case 33 bytes must be transferred per warp, as sums are stored and not individual values of array u. Although this may seem like a serious performance bottleneck, it is not, as eliminating it, by loading one cell per thread, not only does not improves performance but also increases the total run time. This may be explained by the fact that with the uncoalesced access method two successive global reads take place simultaneously, resulting in less data transfers during execution. Moreover, the usage of texture memory also results in correct global memory accesses, however it does not offer a considerable performance improvement.

5.3.2 Kernels of Macroscopic Velocities

For the computation of macroscopic velocities on CPU, four nested *do-loops* take place. The two, outside ones for the spatial grid and two enclosed for the angles and magnitudes. Observing this arrangement, it is greatly similar to a common reduction kernel. To overcome the problem of reduction on two directions of magnitudes and angles, two kernels are being composed. The idea of parallelization is shown in Figure 5.6.

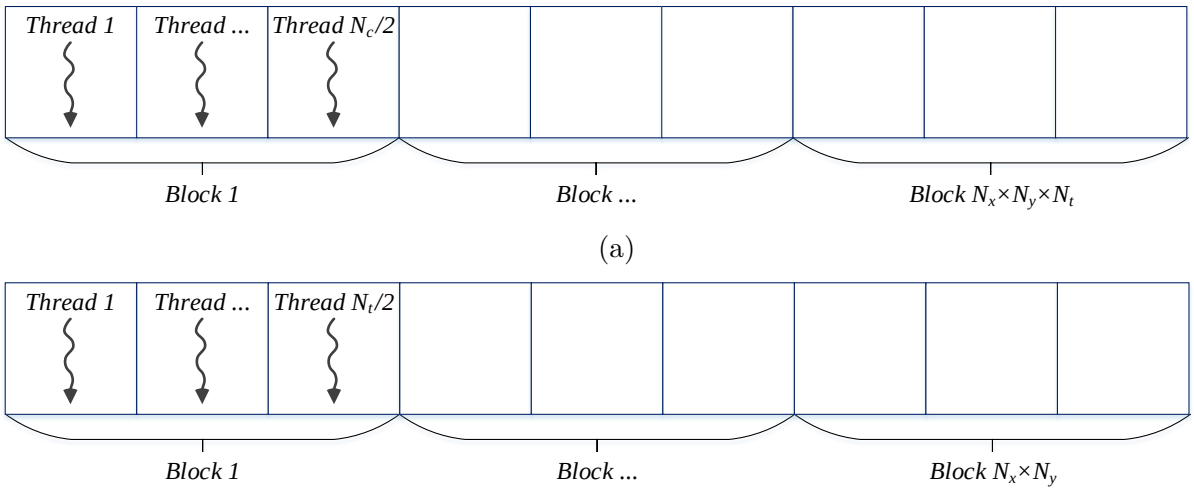


Figure 5.6: Schematic representation of the parallelization strategy of first (a) and second (b) kernel solving for the macroscopic velocities

The first kernel is launched with $Nc/2$ threads per block and $Nt \times Nx \times Ny$ blocks. Inside the kernel a reduction occurs to find the total sum in the direction of magnitudes. Shared memory is utilized in this case, with loading the addition of two elements at once, therein half Nc threads are loaded. Observing the equation for the calculation of one value of shared memory array, two very computationally heavy calculations must occur. Specifically, the two operations with `EXP()` function, although negligible to CPU, are extremely time consuming to GPU calculations. Thus in order to overcome this problem, an array is allocated to global memory that contains the results of `EXP()` function for every magnitude and used accordingly inside the kernel. The equivalent is adjusted accordingly onto the serial code. The results, for a case with $Nc = 128, Nt = 128, Nx = 129, Ny = 129$,

showed approximately 1/3 reduced time during execution on GPU, whereas the CPU time was inconsiderably affected. After the reduction takes place the first element of shared memory tile is transferred to a matrix `sumAng` with dimension $Nt \times Nx \times Ny$, which contains the summations of the reduction for every spatial node and every angle in Nc direction.

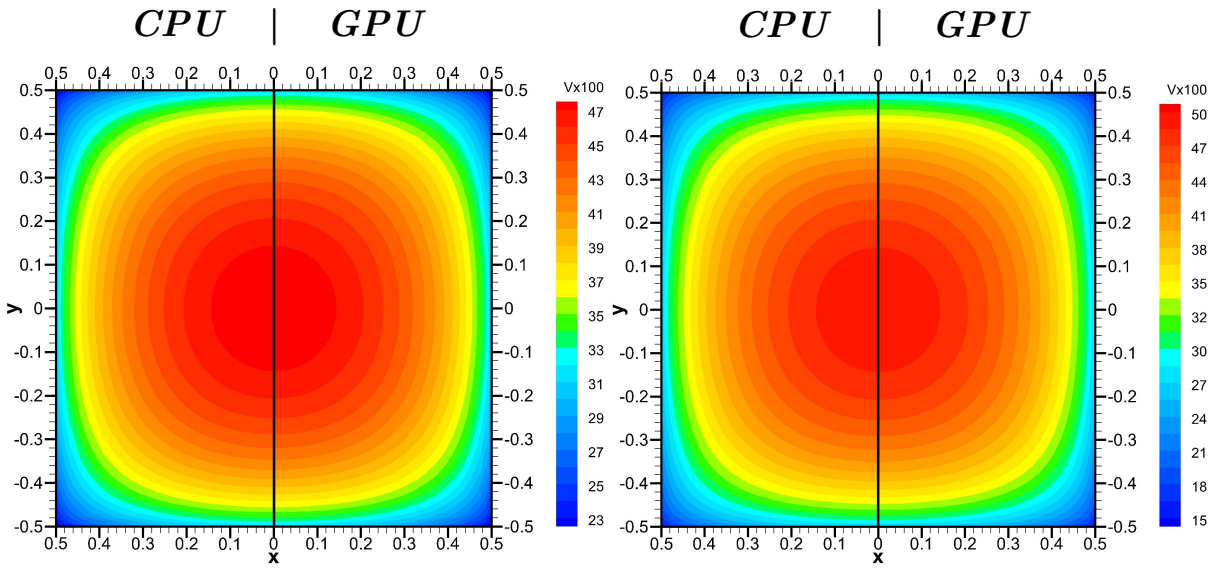
The second kernel is launched with $Nt/2$ threads per block and $Nx \times Ny$ blocks. A reduction take place in the direction of angles Nt, in order to find the new value of the macroscopic velocity of one spatial node. Likewise to the previous kernel, shared memory is utilized with two loads per thread.

Carefulness is required for both kernels with the indices in order to access the linearly fabricated arrays and more importantly the `Y` and `sumAng` arrays. Although the computation of the indices requires several operations –between four and eleven– the kernels' run times did not deteriorate noticeably.

5.4 Results and discussion

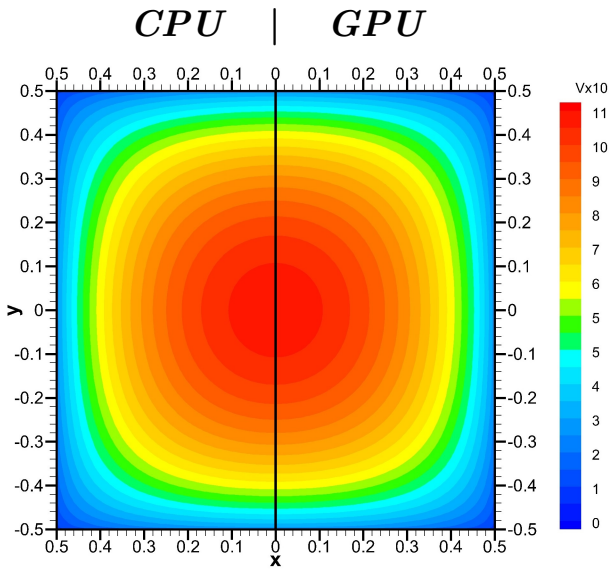
During execution, a relative error for macroscopic velocities, between two successive iterations, of 1.0×10^{-10} for double precision numbers and 1.0×10^{-4} for single precision was held as an upper limit for convergence. The serial code results for the macroscopic velocity profile compared with the parallel ones are shown in Figure 5.7 for values of rarefaction parameter $\delta = 0.1, 1, 10$. The contours of the half CPU velocity profile with the equivalent mirrored GPU profile are in complete agreement with each other, validating the parallel code.

A comparison of the serial and parallel codes was made with various cases. First of all the velocity profile was obtained for a grid of $N_c \times N_t \times N_x \times N_y$ equal to $128 \times 128 \times 257 \times 257$. Secondly, a parametric analysis of the spatial grid was carried out with $N_c = N_t = 64$. Lastly, the effect of N_t angles on performance was studied, for $N_x = N_y = 129$.



(a) $\delta = 0.1$

(b) $\delta = 1$



(c) $\delta = 10$

Figure 5.7: Macroscopic velocity profiles of serial and parallel code for different values of the rarefaction parameter. $N_c = N_t = 128$, $N_x = N_y = 257$ and double precision numbers.

Running the codes to obtaining the results shown in Figure 5.7 for a grid of $N_c \times N_t \times N_x \times N_y$ equal to $128 \times 128 \times 257 \times 257$, the total execution of the parallel code was significantly less than the serial one. In Table 5.1 speed-ups for this dense grid case and for different values of rarefaction parameter as well as different precision are presented.

The overall speed-ups show a tremendous dominance of the parallel code, justifying the choice of Graphical Processing Units for parallelism on this very demanding problem.

Precision	**Rarefaction parameter**	**Speed-up of kernel Y**	**Speed-up of kernel u**	**Overall speed-up**
Double	$\delta = 0.1$	602	29	295
	$\delta = 1$	611	29	298
	$\delta = 10$	602	29	293
Single	$\delta = 0.1$	1082	48	507
	$\delta = 1$	1116	53	535
	$\delta = 10$	1176	57	568

Table 5.1: Speed-ups for a grid of $N_c = N_t = 128$, $N_x = N_y = 257$

Time measurements and metrics of performance were obtained for each kernel separately. In the total run time of the parallel code, all array copies before and after the do-loop for convergence are not included, as their contribution is minimal and thus negligible.

In Figures 5.8 and 5.9 the speed-ups for the calculation of the distribution function and macroscopic velocities are separately presented, as well as the overall succeeded speed-up. The magnitudes and angles were held constant, while the spatial grid of a square cross section in both dimensions was altered equally. In the x-coordinate the results are given according to the parameter $N_s = N_x \times N_y$, while the variables N_x and N_y change uniformly. As the global memory has a maximum capacity of 12 Gb, a rather coarse grid of magnitudes and angles is selected, in order to examine the effect of spatial nodes quantity to the overall speed-up. It is noted that results are only presented here for $\delta = 10$ as the speed-ups do not change at all for $\delta = 0.1$ or 1. The overall resulting speed-up has a maximum of approximately 200 for double precision numbers and 230 for single precision, despite of the coarse $N_c \times N_t$ grid.

The speed-up of the distribution function kernel is substantial, with a maximum of 300 for double and 400 for single precision. However, the overall speed-up is degraded, due to the low speed-up for the computation of macroscopic velocity. The latter is relatively low, because the calculation of u array on host is already significantly lower than the calculation

of `Y`, thus the same performance improvement is not expected when the computation of `u` is parallelized.

Precision	Rarefaction parameter	Iterations
Double	$\delta = 0.1$	10
	$\delta = 1$	29
	$\delta = 10$	330
	$\delta = 100$	17091
Single	$\delta = 0.1$	5
	$\delta = 1$	12
	$\delta = 10$	107
	$\delta = 100$	2598

Table 5.2: Total number of iterations till convergence for different values of rarefaction parameter

In Table 5.2 the total number of iterations required till convergence are shown for different values of the rarefaction parameter. It is interesting to state that since the total number of iterations for $\delta = 100$ is 17091 and speed-ups do not alter for δ, the execution time of the serial code for the most demanding case ($N_c = N_t = 64, N_x = N_y = 577$) can be approximated as follows: The total execution time for $\delta = 10$ for this exact case is 5.4 hours. The ratio of iterations for $\delta = 100$ to $\delta = 10$ is 51.79. Ergo, the approximate run time for $\delta = 100$ is expected to be 279 hours or 11.63 days. The equivalent approximate run time for the parallel code is 7.7 minutes. This difference is of course assumed to be greater for denser grids.

Comparing the global memory version of distribution function kernel with the shared memory version, for the case of $N_c = N_t = 64$, the maximum speed-up is 34 times greater for $\delta = 10$, which one should conclude that is quite insignificant. On the contrary, for a very demanding case, that is a grid of $N_c \times N_t \times N_x \times N_y$ equal to $128 \times 128 \times 257 \times 257$, the speed-up introduced with the shared memory has a tremendous improvement of 100 times increased compared to the version with only global memory utilization. Due to the kernel's structure the aforementioned improvement is expected to be greater for cases with non-square cross sections, where the N_x direction will have more spatial nodes.

In Figures 5.10 and 5.11 the relative time spend on GPU to solve for the distribution

function, time T_Y with a solid bar, and for the macroscopic velocity, time T_u with a blank bar, is shown for double and single precision numbers. The relative time is calculated according to

$$t_r = \frac{t_k}{t_t}, \tag{5.25}$$

where t_r is the relative time of the kernel, $t_r \in [0,1]$, t_k is the run time of the kernel of interest and t_t is the total time of all kernels. The solving of distribution function requires more time, although not significantly, because, as aforementioned, the speed-up of macroscopic velocities greatly suffers.

The memory bandwidth shown in Figure 5.12 and the computational throughput in Figure 5.13 are satisfactory high compared to the upper device limits. Results are shown for the kernel of distribution function and the first kernel of macroscopic velocities. The second kernel of u is not included, as its contribution to the overall times is negligible. Nevertheless, it is noted that the computational bandwidth is approximately the same to first kernel of u and also the kernel's memory bandwidth reaches a maximum value 285 Gb/s for double precision numbers. The line of the bandwidth has the expected curve, whereas the throughput one begins already from an increased value. Due to the structure of the kernel, throughput of less spatial nodes cannot be tested.

Figure 5.14 illustrates the optimal number of blocks, which seems to be 448, in order to hide memory and operation latency. In other words, the GPU, for this specific number of blocks, is filled with enough "waves" of blocks to hide memory transfers, while the cycles to complete the operations are kept to a minimal. The latter statement is explained by the degradation of speed-up after loading more than 448 blocks.

Likewise, in Figure 5.15 the relative time is shown versus the number of angles N_t. The time spend to calculate the distribution function seems to be equivalent to the other two for the macroscopic velocities for all angles. This happens, because the first kernel of u is launched with $N_x \times N_y \times N_t$ blocks and thus increasing the N_t parameter the GPU has

a lot more waves of warps to run for this kernel. Contrariwise, it is beneficial to increase the parameter N_t for the kernel of `Y`, as even for the largest value, shown in Figure 5.15, the number of blocks for this kernel is fairly small, with sufficient waves of warps to hide latencies. This effect can be also observed on the overall speed-up in Figure 5.14 as both curves contribute equally to the middle one.

The succeeded memory bandwidth, presented in Figure 5.16, is higher compared to the previous examined case, as more threads per grid are launched for all kernels. With a hardware maximum of 547.6 GB/s, the top bandwidth for the kernel of distribution function, approximately half the limit, shows satisfactory data handling via the kernel and achieved parallelism. The computational throughput, shown in Figure 5.17, has the expected ascending curve for the kernel calculating `Y`, whereas for kernel of macroscopic velocities it is fairly constant. The cause of the latter behavior is that although N_t is low at the beginning, the first kernel is loaded with many blocks, $N_x \times N_y \times N_t$, where parameters N_x and N_y have already high values for this case. As a result, the computational throughput and memory bandwidth for the kernel of macroscopic velocities has already reached its limit, even for a low number of angles.

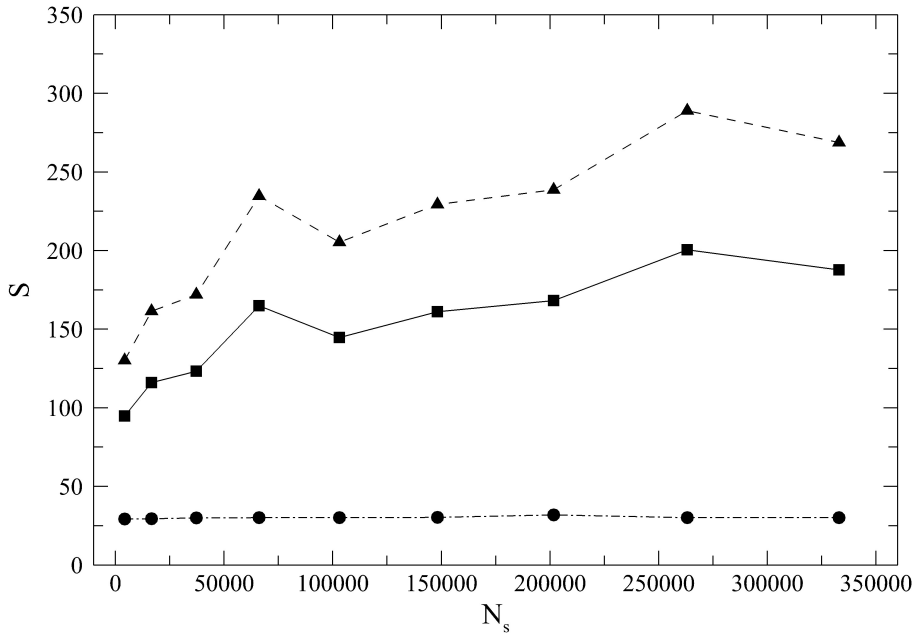


Figure 5.8: Speed-up, S, versus the number of cells in the physical space. Dashed line with triangle symbols for distribution function kernel; dashed-dot line with circle symbols for macroscopic velocity kernels; solid line with square symbols for the overall speed-up. $\delta = 10$, $N_c = N_t = 64$ and double precision.

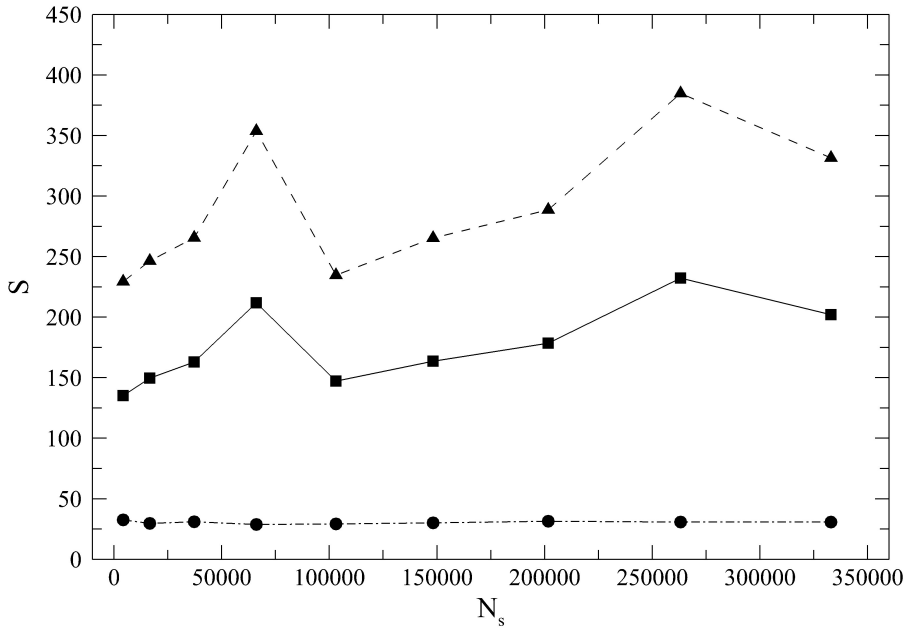


Figure 5.9: Speed-up, S, versus the number of cells in the physical space. Dashed line with triangle symbols for distribution function kernel; dashed-dot line with circle symbols for macroscopic velocity kernels; solid line with square symbols for the overall speed-up. $\delta = 10$, $N_c = N_t = 64$ and single precision.

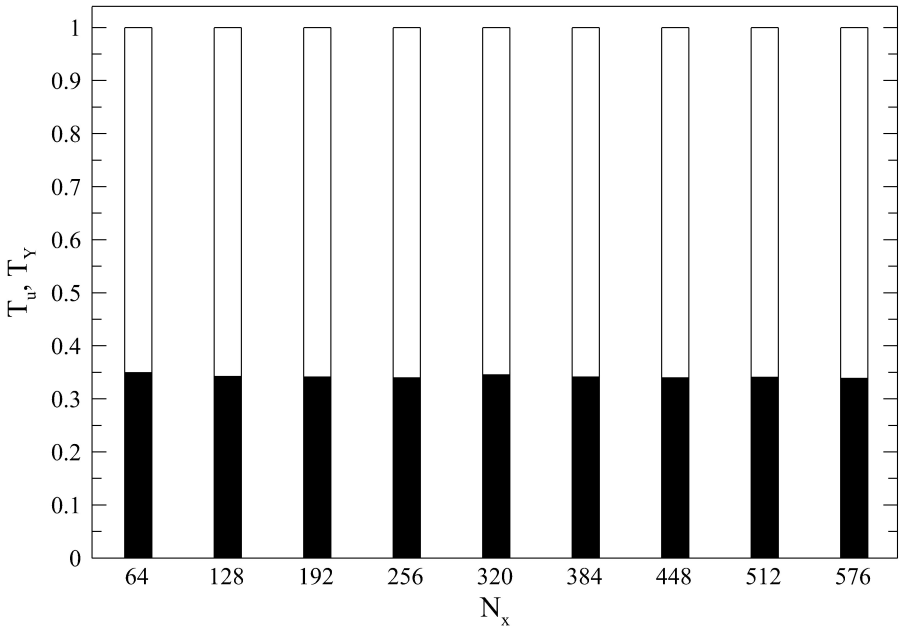


Figure 5.10: Relative time spend on macroscopic velocity kernel, with solid bar, and on distribution function kernel, blank bar. $\delta = 10$, $N_c = N_t = 64$ and double precision.

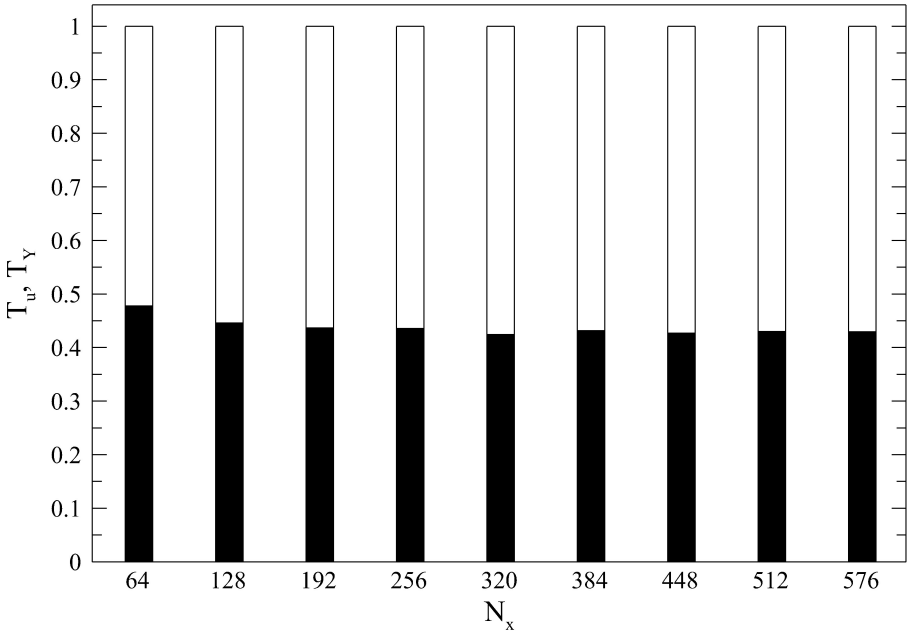


Figure 5.11: Relative time spend on macroscopic velocity kernel, with solid bar, and on distribution function kernel, blank bar. $\delta = 10$, $N_c = N_t = 64$ and single precision.

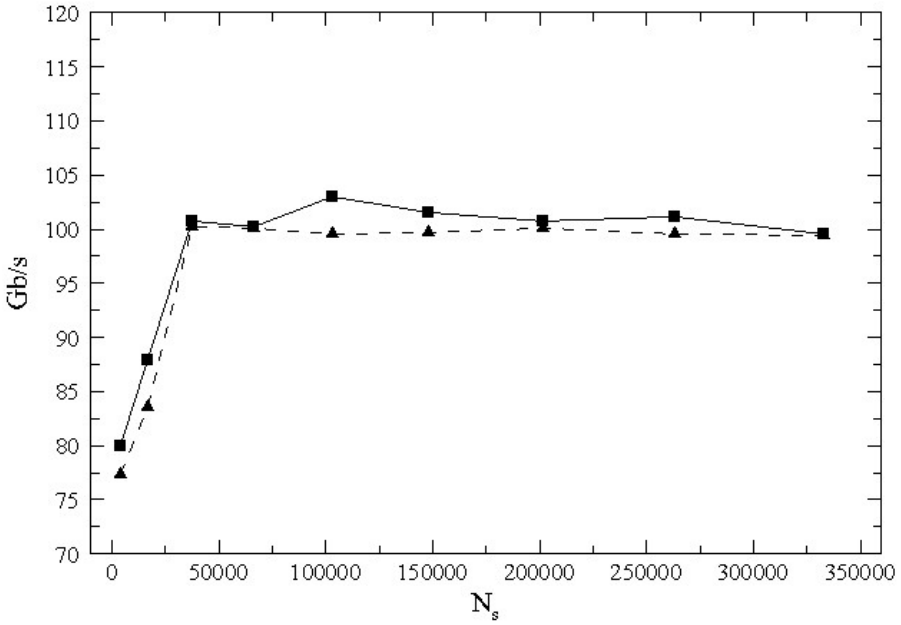


Figure 5.12: Memory bandwidth versus the number of cells in the physical space. Solid line with square symbols for distribution function kernel; dashed line with triangle symbols for the first macroscopic velocity kernel. $N_c = N_t = 64$ and double precision.

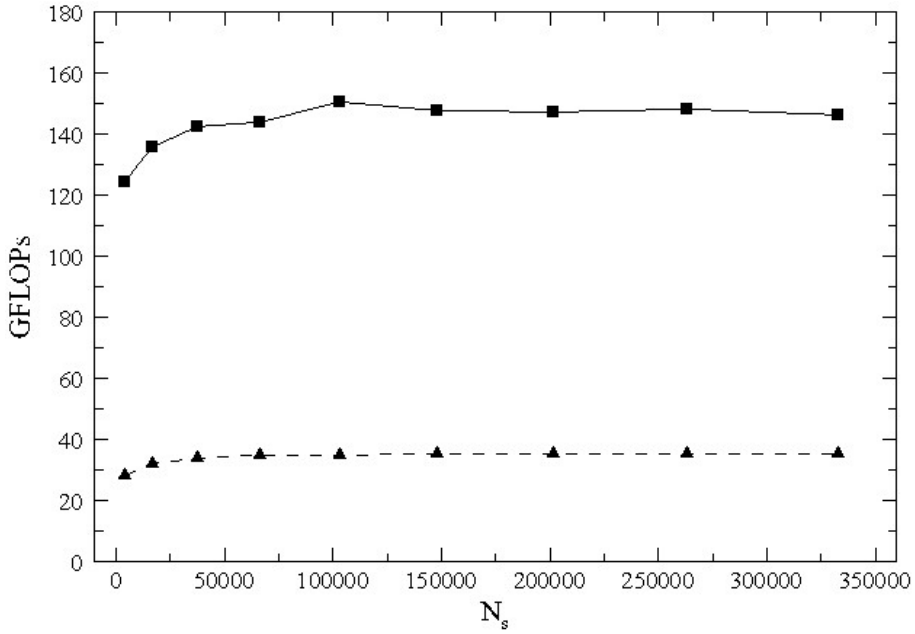


Figure 5.13: Computational throughput versus the number of cells in the physical space. Solid line with square symbols for distribution function kernel; dashed line with triangle symbols for the first macroscopic velocity kernel. $N_c = N_t = 64$ and double precision.

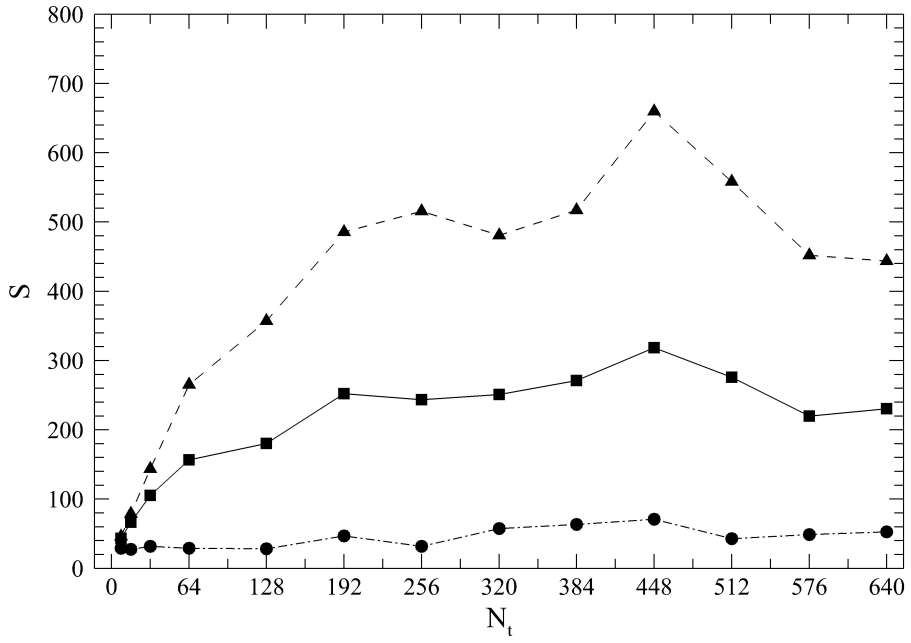


Figure 5.14: Speed-up, S, versus the number of angles. Dashed line with triangle symbols for distribution function kernel; dashed-dot line with circle symbols for macroscopic velocity kernels; solid line with square symbols for the overall speed-up. $\delta = 1$, $N_c = 128$, $N_x = N_y = 129$ and double precision.

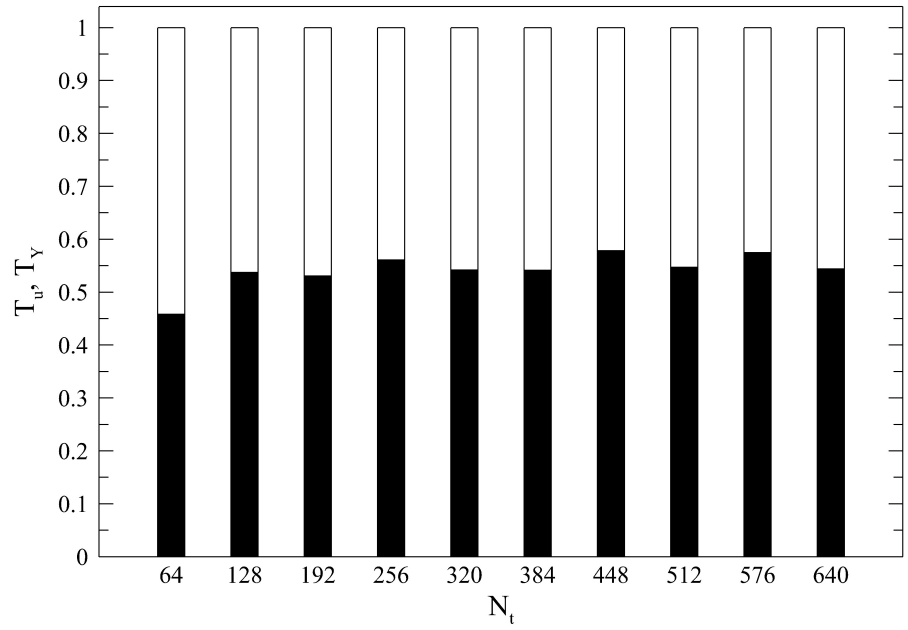


Figure 5.15: Relative time spend on macroscopic velocity kernel, with solid bar, and on distribution function kernel, blank bar. $\delta = 1$, $N_c = 128$, $N_x = N_y = 129$ and double precision.

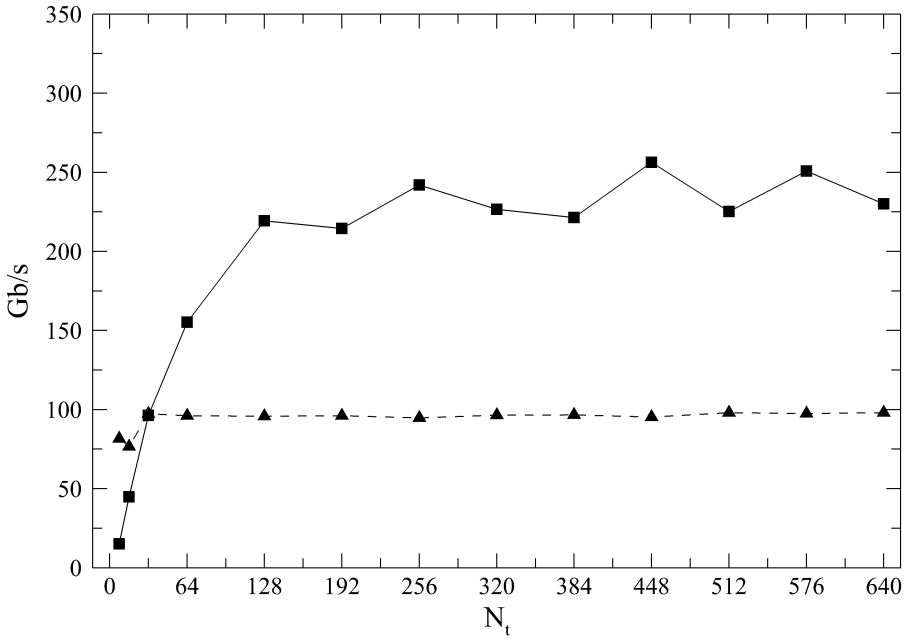


Figure 5.16: Memory bandwidth versus the number of angles. Solid line with square symbols for distribution function kernel; dashed line with triangle symbols for the first macroscopic velocity kernel. $N_c = 128$, $N_x = N_y = 129$ and double precision.

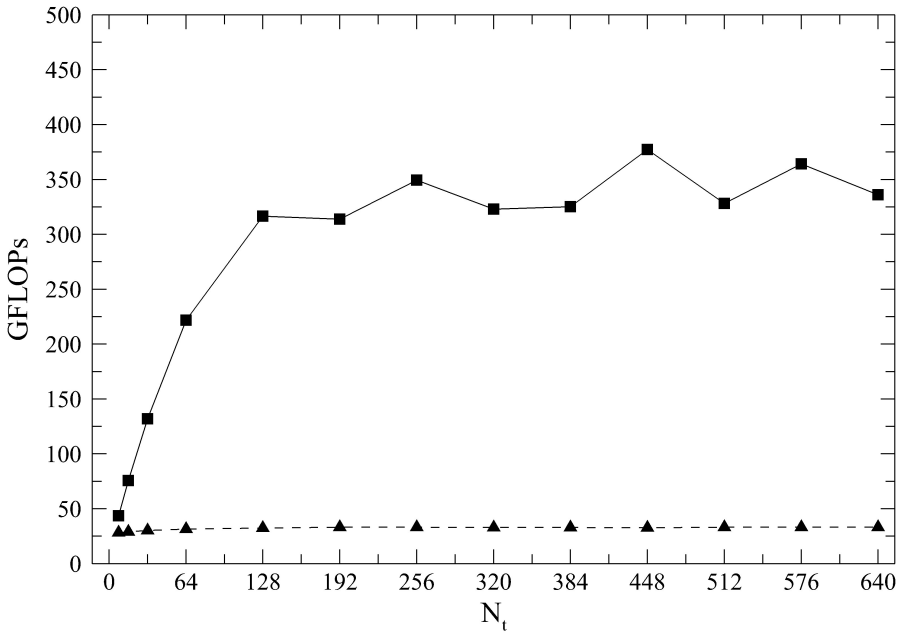


Figure 5.17: Computational throughput versus the number of angles. Solid line with square symbols for distribution function kernel; dashed line with triangle symbols for the first macroscopic velocity kernel. $N_c = 128$, $N_x = N_y = 129$ and double precision.

Concluding remarks

In this diploma thesis the implementation of benchmark mechanical engineering problems on Graphical Processing Units was examined. Using the parallelism technique of CUDA Application Programming Interface it was able to accelerate the serial codes execution significantly.

In chapter 3, a problem of heat transfer through an L-shaped fin was studied. The shape, geometry and size of the fin finds great application on cooling fins for computer parts. The serial execution suffers significantly of computational time, as in order to obtain the temperature profile the code has to perform many iterations to converge to a solution. Applying CUDA Fortran the serial code was accelerated by up to 410 times.

In chapter 4, the flow of a rarefied gas between two parallel plates was examined. As low pressure difference is applied on both sides of the plates, the flow is laminar and is known in the literature as Poiseuille flow. The plates is assumed to be of infinite width, thus the problem can be reduced to an one-dimensional. Using CUDA Fortran the parallel code offered a maximum speed-up of 46 for the most demanding case.

In chapter 5, an extension of the previous chapter, the Poiseuille flow of a rarefied gas through a duct was studied. The length of the duct is assumed to be much greater than the two other dimensions. As the duct's margins are included for the solution, the flow is reduced to a two-dimensional. The serial code was accelerated by a maximum of 298 for double precision and 568 for single precision numbers.

Continuation and further development of the present research may be performed in the following directions relating to the physical problem examination:

- Different fin geometries can be studied, as well as different boundary conditions,

because the extension of the code is straight forward. In that direction, the shape of the fin can be altered in order to fit more complex and modern fins used currently in cooling applications. A parametric analysis of the effect different parameters have on cooling capabilities, such as air velocity or material thermal conductivity, can be also performed.

- The parallel code of a rarefied gas subjected to a laminar, Poiseuille flow could be extended to more complex geometries, such as the one presented in [65], which studies the gas flow including the effect both vessels have on each side of the duct. The structure of the problem is divided into three independently solvable code segments, which can be replaced by three concurrently executed kernels, increasing even more the succeeded parallelism. Moreover, the parallel codes of one- and two-dimensional flow of rarefied gases can be applied to polyatomic or mixtures of monoatomic gases. This can be succeeded by altering the necessary equations, thoroughly described in [66].

- The developed parallel code of all studied problems can be easily expanded to include solving for three-dimensional geometries. In heat transfer application a different nodal structure is required as well as an additional *do-loop*. For the Poiseuille flow, following the two-dimensional extension from the one-dimensional, the development of the parallel code for a three-dimensional flow is straight forward.

Relevant to parallel computing using CUDA, tasks for future work could be:

- The utilization of unified memory on all of the aforementioned problems. Unified Memory is a single memory address space accessible from any processor in a system [67]. This hardware/software technology allows applications to allocate data that can be read or written from code running on either CPUs or GPUs. This would eliminate the need to transfer data from host DRAM to device global memory.

- Moreover, the speed-up would show a tremendous increase by utilizing a GPU cluster. Minimal alterations must be maid inside each kernel to assign blocks on different

devices, in order to be executed concurrently. In that direction more threads, thus problem's nodes, can be utilized, in favor of obtaining more accurate solutions, as the only limiting factor is the capacity of global memory.

- Finally, hybrid models offer the optimal choice, as usually a segment of a code may be more appropriate to be parallelized by another parallel computing model. Various hybrid models could be examined, such as OpenMP/CUDA, such as the one used in [21], or MPI/CUDA, similarly to [68], or even both of the aforementioned API on a unified hybrid model, as the developed one by [69], offering a huge assistant on code execution performance.

Appendices

Appendix A: Program execution

All developed codes were compiled using the Portland Group® Inc. (PGI) compilers, which is a powerful set of commercially available Fortran compilers for high-performance computing systems. Using these software development tools it is able to produce executable code, that is written according to CUDA Fortran Application Programming Interface.

Both serial and parallel codes have the suffix `.cuf`, in order to be compatible with the compiler. The compiling command for a serial code, named `sTest.cuf` is:

- `$ pgf90 -Mpreprocess -DDOUBLE sTest.cuf`: *for Linux operating system,*
- `$ pgf90.exe -Mpreprocess -DDOUBLE sTest.cuf`: *for Windows operating system.*

The first part of the above command refers to the high-level programming language that the code is written to, with a small modification to fit each of the two utilized operating systems. It is important to state here that no time or performance differences were observed using other commercially available compilers for Fortran language and that is the reason why the serial code was compiled with the PGI compilers. The second two flags, `-Mpreprocess` and `-DDOUBLE` refer to the `precisionFpKind` module at the beginning of each code, which is cited below, that defines whether single or double precision numbers will be used. Compiling with the flags `-Mpreprocess -DDOUBLE` if the file has a ".cuf" extension or `-DDOUBLE` for a ".CUF" extension, the results will be on double precision. Alternatively if the `-DDOUBLE` flag is excluded during compiling then the solution will output single precision results. Inside the main code the initialization of a real variable

must have the format `real(fpKind)::` and the variable names; in an operation with real numbers, the suffix `_fpKind` must be used after the number (for example `1.0_fpKind`).

```
1:  module precisionFpKind
2:
3:      integer, parameter ::singlePrecision =kind(0.0)
4:      integer, parameter ::doublePrecision =kind(0.0d0)
5:
6:      ! To compile with double precision include the flags:
7:      !  -Mpreprocess -DDOUBLE (if the file has .cuf extension)
8:      !  -DDOUBLE           (if the file has .CUF extension)
9:
10:     #ifdef DOUBLE
11:         integer, parameter ::fpKind =doublePrecision
12:     #else
13:         integer, parameter ::fpKind =singlePrecision
14:     #endif
15:
16: end module precisionFpKind
```

Listing A.1.1: Module that defines the usage single or double precision numbers during compiling.

A parallel code, named as `pTest.cuf`, is compiled using the command:

- `$ pgf90 -ta=tesla:cc50 -Mpreprocess -DDOUBLE pTest.cuf`: *for Linux operating system,*
- `$ pgf90.exe -ta=tesla:cc50 -Mpreprocess -DDOUBLE pTest.cuf`: *for Windows operating system.*

The additional flag refers to the architecture of the GPU and is essential to be included during compiling, in order to produce the optimal executable for each device. The

`-ta=tesla:cc50` refers to the NVIDIA Quadro M2200 GPU, whereas when compiling on the NVIDIA Titan Xp GPU the correct flag is the `-ta=tesla:cc60`.

In order to run the executable of the serial or parallel code, the commands that should be typed in the terminal are:

- `$ ./sTest`: *for Linux operating system,*
- `$ sTest.exe`: *for Windows operating system.*

Appendix B: PGI Profiler

Metric name	Description
achieved_occupancy	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
inst_per_warp	Average number of instructions executed by each warp
sm_efficiency	The percentage of time at least one warp is active on a multiprocessor
branch_efficiency	Ratio of non-divergent branches to total branches expressed as percentage
warp_execution_efficiency	Ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor expressed as percentage
local_load_transactions	Number of local memory load transactions
local_store_transactions	Number of local memory store transactions
local_memory_overhead	Ratio of local memory traffic to total memory traffic between the L1 and L2 caches expressed as percentage
shared_load_throughput	Shared memory load throughput
shared_store_throughput	Shared memory store throughput
shared_efficiency	Ratio of requested shared memory throughput to required shared memory throughput expressed as percentage

Metric name	**Description**
shared_utilization	The utilization level of the shared memory relative to peak utilization on a scale of 0 to 10
dram_read_throughput	Device memory read throughput
dram_write_throughput	Device memory write throughput
flops_sp	Single-precision floating point operations executed
flops_sp_add	Single-precision floating point add operations executed
flops_sp_mul	Single-precision floating point multiply operations executed
flops_sp_fma	Single-precision floating point multiply-accumulate operations executed
flops_dp	Double-precision floating point operations executed
flops_dp_add	Double-precision floating point add operations Multi-context executed
flops_dp_mul	Double-precision floating point multiply operations executed
flops_dp_fma	Double-precision floating point multiply-accumulate operations executed
stall_data_request	Percentage of stalls occurring because a memory operation cannot be performed due to the required resources not being available or fully utilized, or because too many requests of a given type are outstanding
stall_sync	Percentage of stalls occurring because the warp is blocked at a `syncthreads()` call

Table B.1: Useful PGI command-line profiler metric flags [46]

PGI Visual profiler or PGI command-line profiler are excellent tools to obtain various important metrics, in order to evaluate a kernel's performance. A short description of the steps one should follow to collect the metrics is presented subsequently. Instruction and results are shown, both for the visual and command-line profiler.

Most essential metrics of performance are:

- occupancy
- memory bandwidth
- computational throughput
- global memory coalesced access
- shared memory bank conflict

The results were obtained by examining a kernel that adds two arrays, `a` and `b`, of same size together. The code of the kernel is given below. Since arrays are one-dimensional, the grid and blocks were also one-dimensional. Kernel is invoked with 256 threads per block and as many blocks to fill the size of the arrays. The total number of threads, equal to the size of arrays, is 204800. The NVIDIA Quadro M2200 was used to retrieve the results.

```
1:  attributes(global) subroutine additionKernel(a, b, x)
2:      implicit none
3:      real(8) ::a(:), b(:)
4:      real(8), value ::x
5:      integer ::i, n
6:
7:      n = size(a)
8:      i = blockDim%x * (blockIdx%x - 1) + threadIdx%x
9:
10:     if (i <= n) a(i) = a(i) + x*b(i)
11: end subroutine additionKernel
```

Listing B.1.2: Kernel to add two arrays.

To initiate a new session with Visual profiler one should write the command `pgprof` in the terminal to open the profiler window. In the main ribbon clicking on *"File"* and then

"New session" opens a window. There in *"File"* browse to the executable and press *"open"*. After pressing *"Next"* one has the ability to choose the profiling and timeline options. Under the *"Profiling options"* the preferences *"Start execution with profiling enabled"*, *"Enable concurrent kernel profiling"*, *"Enable CUDA API tracing in the timeline"* should be ticked. Moreover in *"Timeline Options"*, under *"Enable timelines in session view"* the option *"All"* should be ticked. After click *"Finish"*. The timeline is now being generated.

Firstly, the achieved occupancy is obtained. The command for occupancy from the command-line profiler is:

```
pgprof --metrics achieved_occupancy ./additionKernel
```

The output is shown in Figure B.1.

```
Invocations          Metric Name     Metric Description          Min      Max      Avg
Device "Quadro M2200 (0)"
    Kernel: mathops_saxpy_
          1      achieved_occupancy   Achieved Occupancy     0.872715    0.872715    0.872715
```

Figure B.1: Command-line profiler output for achieved occupancy (Modified to fit page).

Same achieved occupancy can be seen from Visual profiler, as shown in Figure B.2. The occupancy position output is highlighted with the blue ellipse, in the Properties window.

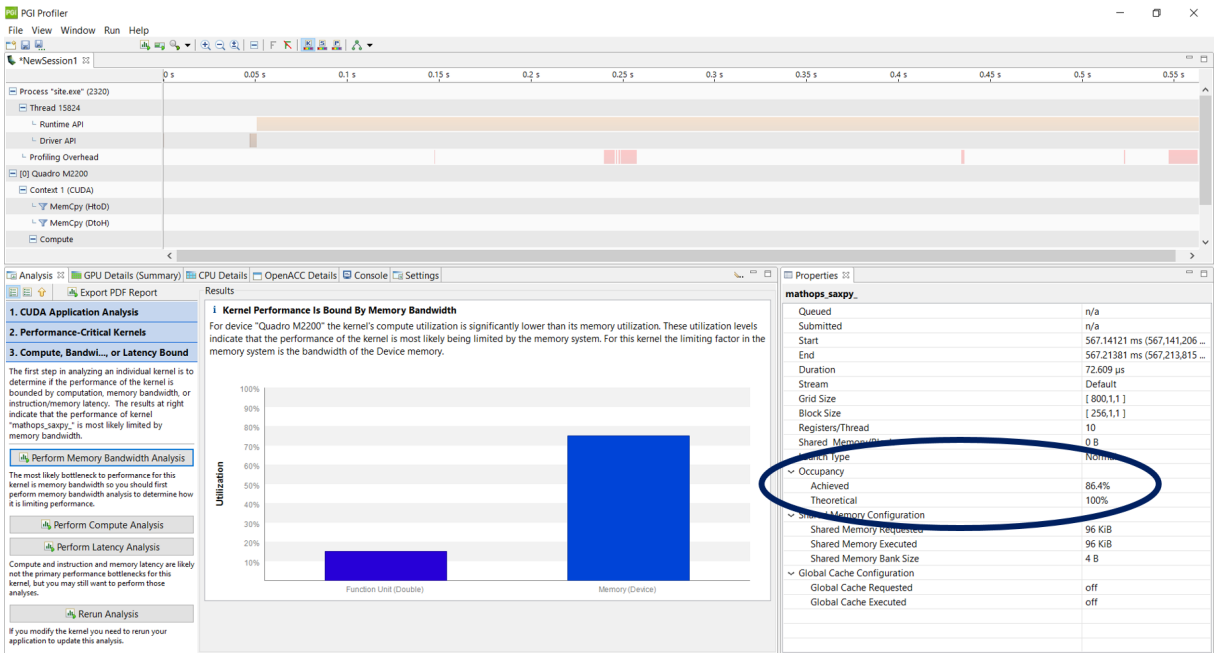


Figure B.2: Screenshot of PGI Visual Profiler for occupancy calculation.

As described in Section 2.6.5, memory bandwidth is a crucial performance metric, especially for memory-bound kernels. The output for the command-line profiler is:

```
pgprof --metrics dram_read_throughput,dram_write_throughput ./additionKernel
```

In order to generate the correct result, the PGI profiler repeats the measurements several times and outputs a minimum, maximum and average value. In Figure B.3 the actual output is presented.

```
Invocations      Metric Name    Metric Description      Min      Max      Avg
Device "Quadro M2200 (0)"
    Kernel: mathops_saxpy_
 1   dram_read_throughput  Device Memory Read Throughput  40.034GB/s  40.034GB/s  40.034GB/s
 1   dram_write_throughput  Device Memory Write Throughput  20.014GB/s  20.014GB/s  20.014GB/s
```

Figure B.3: Command-line profiler output for bandwidth (Modified to fit page).

To generate the same output from the Visual profiler one should select *"Examine Individual Kernels"*, from the *"Analysis window"*, *"1.CUDA Application Analysis"*. Then select the desired kernel from the *"Kernel Optimization Properties"* frame, under *"Results"* window. Afterwards, select *"Perform Kernel Analysis"* from *"2. Performance-*

Critical Kernels" window. this will execute the application one or more times to generate the timeline. Pressing *"Perform Memory Bandwidth Analysis"*, under *"3. Compute, Bandwidth, or Latency Bound"* will output the memory bandwidth results. Usually the memory bandwidth of interest is the bandwidth of device DRAM transfers during kernel execution, which is located inside *"Device Memory"* frame, as shown in Figure B.4, as shown from the blue ellipse.

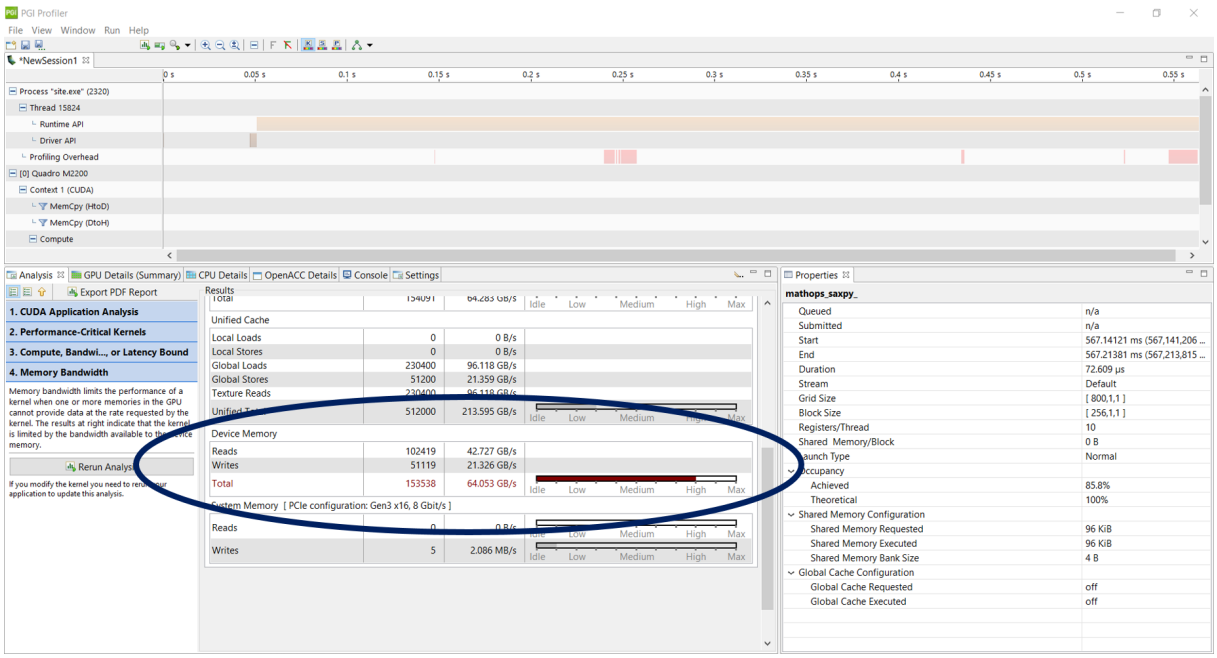


Figure B.4: Screenshot of PGI Visual Profiler for occupancy calculation.

For compute-bound kernels the computational bandwidth is a good estimation of performance when tested on the same device and is compared with other versions of kernels that produce the same output. The GFLOPs are calculated according to Equation 2.8. The execution time of the kernel is generated by simply running the command `pgprof ./additionKernel`, whereas the total number of FLOPs of all threads (i.e. the term $n_t \times$ FLOPs) by the command:

```
pgprof --metrics flops_sp,flops_dp ./additionKernel
```

The output of the above command is shown in Figure B.5. The command `flops_sp` gives the single precision flops, whereas the `flops_dp` the double precision flops. The total number of flops is generated by adding the single and double precision flops.

```
Invocations      Metric Name    Metric Description      Min      Max      Avg
Device "Quadro M2200 (0)"
 1  flop_count_dp  Floating Point Operations(Double Precision)  409600  409600  409600
 1  flop_count_sp  Floating Point Operations(Single Precision)  0       0       0
```

Figure B.5: Command-line profiler output for computational throughput (Modified to fit page).

Using the visual profiler, the total FLOPs can be shown in tab *"GPU Details (Summary)"*, as shown in Figure B.6 by the blue ellipse. In order for the *"Floating Point Operations(Double precision)"* and *"Floating Point Operations (Single precision)"* columns to be generated, one should click on *"Run"* option in the top ribbon, then *"Configure Metrics and Events"* and finally in the *"Metrics"* tab to select and tick the *"Floating Point Operations (Double precision)"* and *"Floating Point Operations(Single precision)"* options under *"Instruction"*. Afterwards the data will be recollected and the two additional columns will be included.

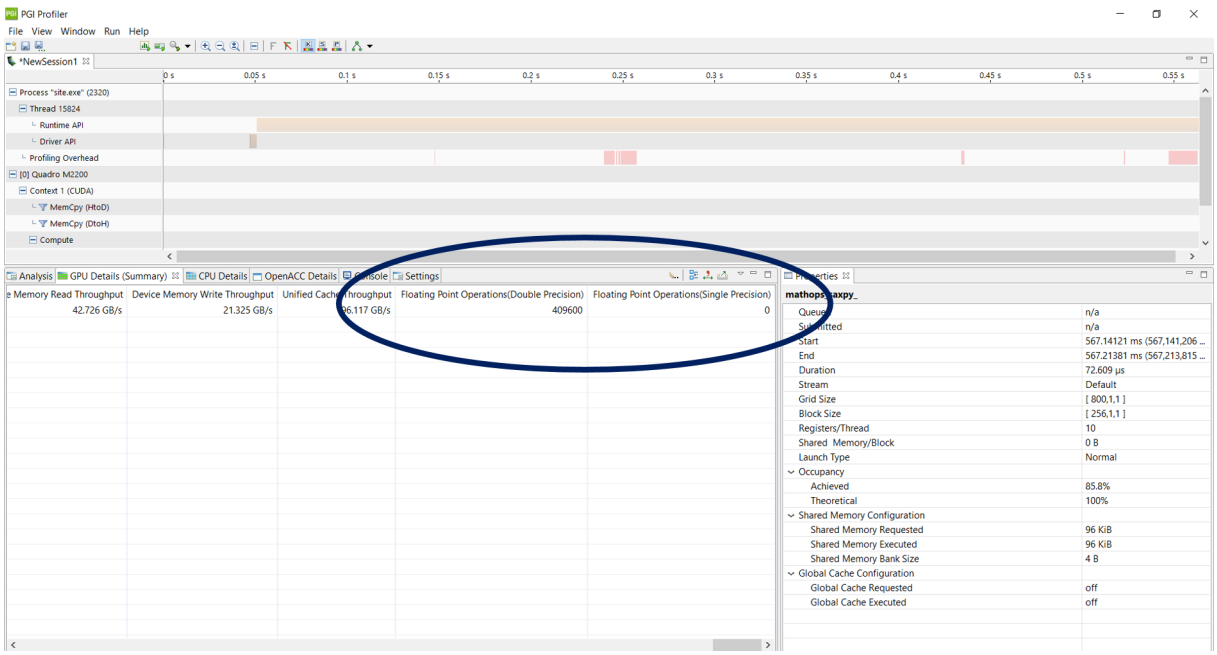


Figure B.6: Screenshot of PGI Visual Profiler for occupancy calculation.

To achieve ultimate performance, coalesced access to global memory should be confirmed. This can easily be accomplished via the Visual Profiler by clicking the *"Unguided*

Analysis", after having selected the preferred kernel, as indicated by the red circle in Figure B.7. Thereupon, in *"Analysis"* tab selecting the *"Global Memory Access Pattern"* option will output if an access to global memory problem exists, where the blue ellipse points.

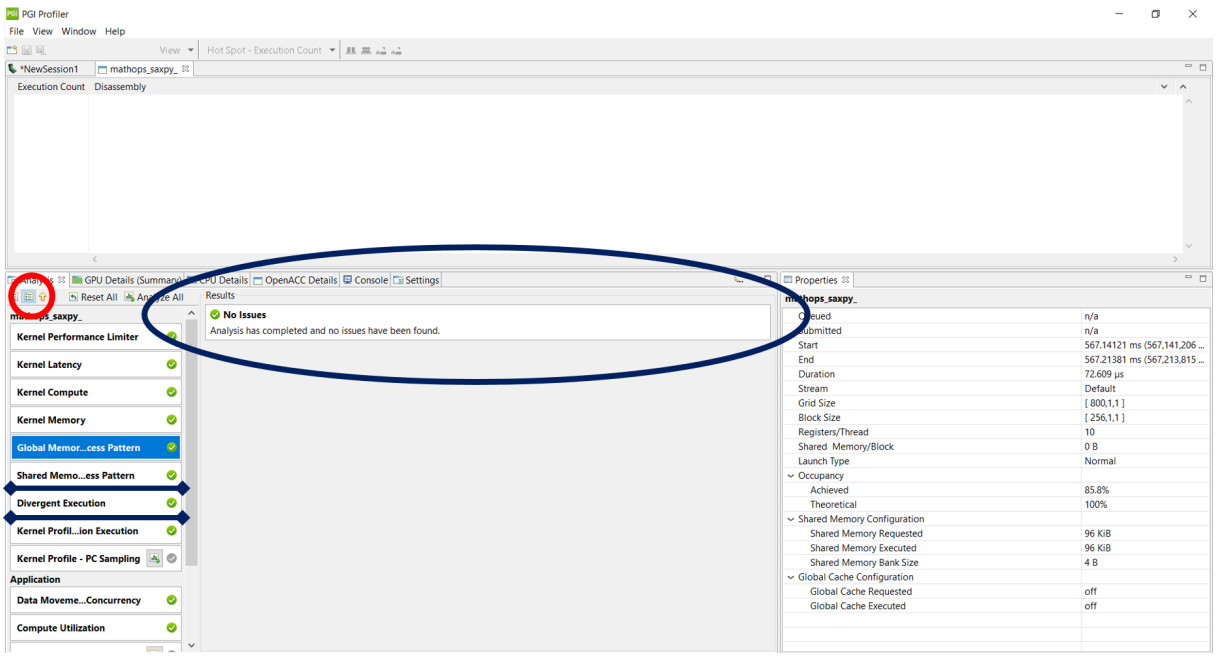


Figure B.7: Screenshot of PGI Visual Profiler for occupancy calculation.

To check for bank conflicts in shared memory the option *"Shared Memory Access Pattern"* should be selected, which is located underneath the *"Global Memory Access Pattern"*. In the same place where the blue ellipse shows in Figure B.7, a relevant message will be shown in case a problem accessing shared memory exists.

Eventually, to check for warp divergence one can select the option *"Divergent Execution"* underneath the *"Shared Memory Access Pattern"*. This will inform the user for any divergent warps. To output the branch efficiency via the command-line profiler one should execute the command:

```
pgprof --metrics branch_efficiency ./additionKernel
```

The actual output is displayed in Figure B.8.

```
Invocations     Metric Name   Metric Description     Min      Max      Avg
Device "Quadro M2200 (0)"
 1        branch_efficiency   Branch Efficiency    100.00%  100.00%  100.00%
```

Figure B.8: Command-line profiler output for branch efficiency (Modified to fit page).

To include line info when running the Visual profiler, for example to inform the user about the actual line in the source code where a *"Global Memory Access Pattern"* or a *"Shared Memory Access Pattern"* problem is encountered, the flag `-Mcuda=lineinfo` must be included during compiling. This will display the line where a problem is met, exactly where the blue circle in Figure B.7 aims.

Appendix C: Reduction scheme

The computation of a reduction scheme on GPUs can occur via a kernel of high performance. Carefulness is required during development of this kernel in order to avoid performance bottlenecks. In Figure C.1 the concept of reduction is schematically presented [70].

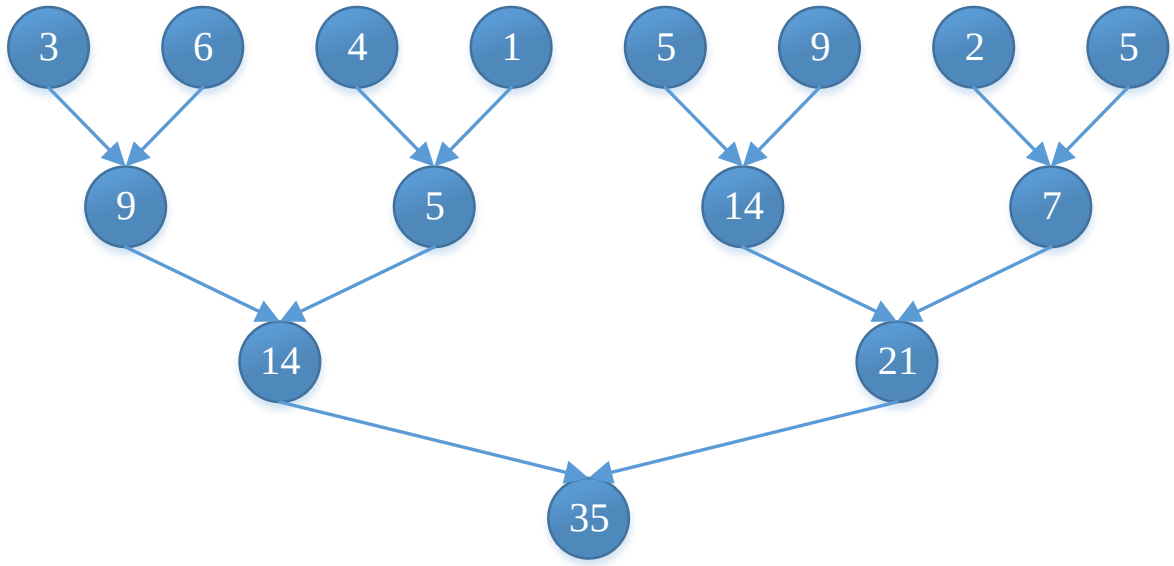


Figure C.1: Schematic representation of the reduction scheme.

Specifically, according to this technique all elements must be compared with each other. Therefore, the reduction compares two elements at a time till all are compared with each other and finally one value is obtained. This is called a *tree-reduction*. The most common reduction operation is computing the sum of a large array of values. Other reduction operations that are often encountered are the computation of the minimum or maximum value of an array.

What is prefered in a reduction kernel is to have all active threads in as few warps as is possible, in order to minimize divergence. This can be achieved by storing the result of one stage of the reduction so that all the active threads for the next stage are contiguous [47]. These two problems are schematically presented in Figures C.2 and C.3.

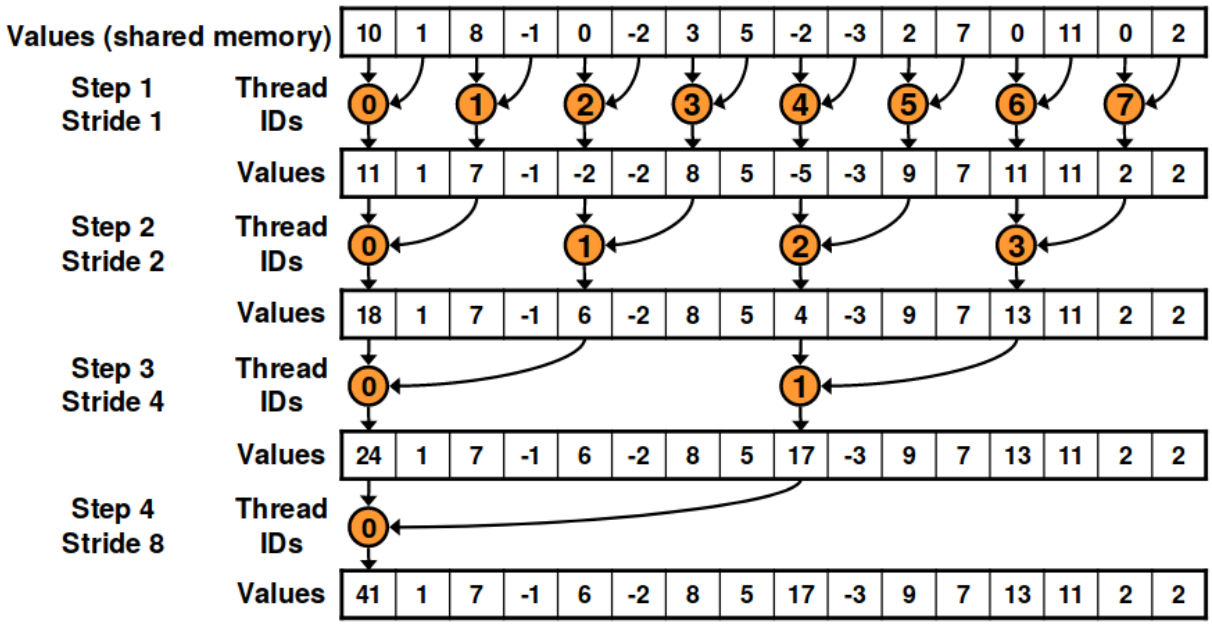


Figure C.2: Parallel reduction with interleaved addressing [70].

In Figure C.2 a case of 16 elements is analyzed, assuming a block with 16 threads for illustrative purposes. Of course in reality many more threads in a block are required, in order to hide latencies. After the values are loaded in shared memory, each active thread at step M, $M = 1, ..., log_2(N)$ will sum its value to the one with stride 2^{M-1}. A careful examination of Figure C.2 shows that improvements can be made. Kernels performance is limited due to thread divergence. For cases where a large number of threads per block are used, a warp of threads in the latter stages of the reduction may have only one active thread. What would be desirable is to have all active threads in as few warps as possible in order to minimize thread divergence. This can be achieved by storing the result of one stage of the reduction so that all the active threads for the next stage are contiguous. This solution is presented in Figure C.3. The code of sequential addressing reduction is cited below:

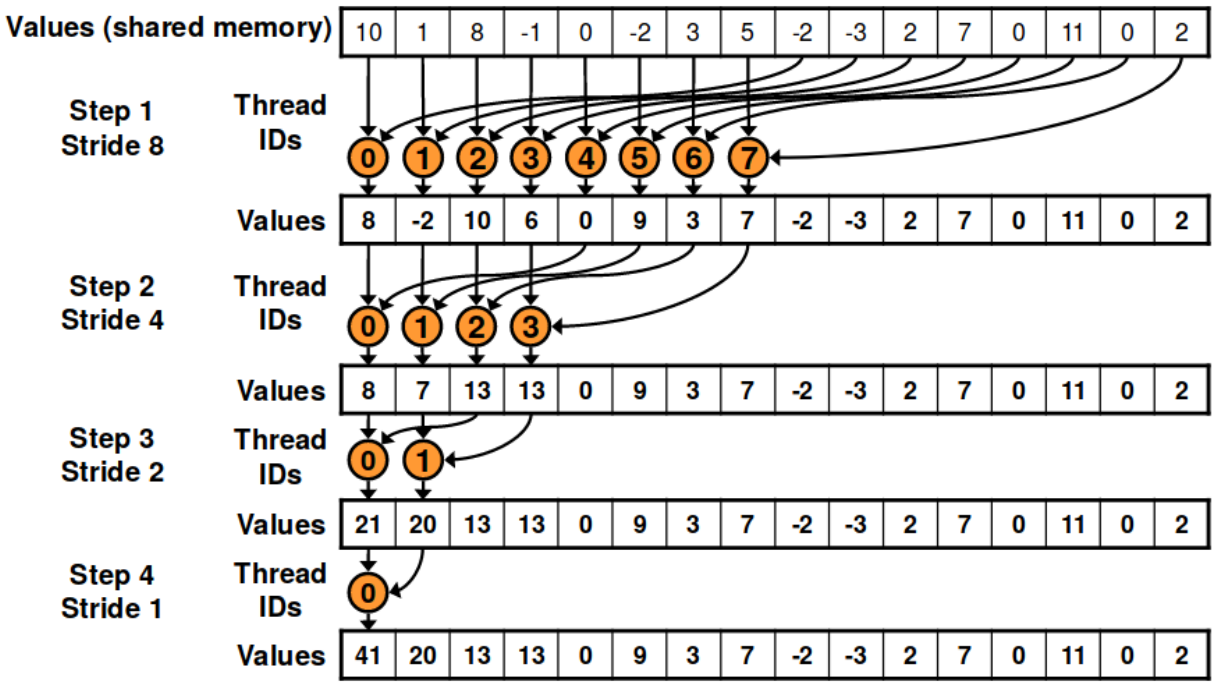


Figure C.3: Parallel Reduction with sequential addressing [70].

```
1:  attributes(global) subroutine residual(partial, total)
2:
3:      real(fpKind), device ::partial(256)
4:      real(fpKind), shared ::psum(*)
5:      integer ::total
6:      integer ::index, inext
7:
8:      index = threadIdx%x
9:
10:     psum(index) =partial(index)
11:
12:     call syncthreads()
13:
14:     inext = blockDim%x / 2
```

```
15:     do while (inext >= 1)
16:         if (index <= inext) psum(index) =psum(index) +psum(index + inext)
17:         inext = inext / 2
18:         call syncthreads()
19:     end do
20:
21:     if (index ==1) total = psum(1)
22:
23: end subroutine residual
```

Listing C.1.3: Parallel reduction with sequential addressing code.

Finally, the implementation of reduction scheme on GPU requires two different levels, if the total elements for reduction exceed the limit of threads per block. The two levels are schematically presented in Figure C.4. First kernel, of level 0, outputs the one summation value of each block and stores them into an array. Afterwards, a second kernel can be developed that takes the summation of every block of level 0, which are stored into a separate array, and adds all of them together. The structure of the second kernel is the same as the first, with only difference the data transfers to and from the shared memory array.

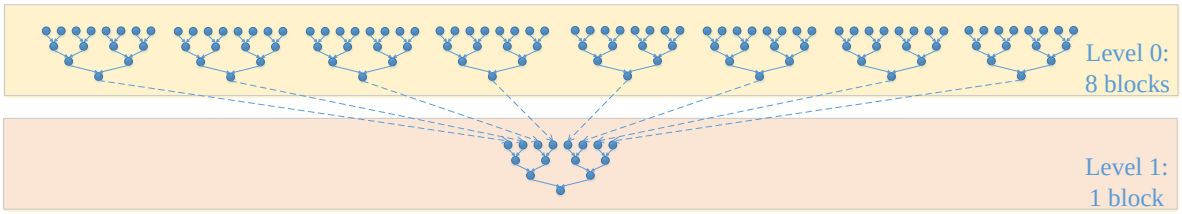


Figure C.4: Implementation of reduction scheme on GPUs.

Bibliography

[1] Gordon E Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998. ISSN 0018-9219.

[2] Nvidia. CUDA C Programming Guide. Technical report, 2018.

[3] Thaddeus D Ladd, Fedor Jelezko, Raymond Laflamme, Yasunobu Nakamura, Christopher Monroe, and Jeremy Lloyd O'Brien. Quantum computers. *Nature*, 464(7285):45, 2010. ISSN 1476-4687.

[4] G Massimo Palma, Kalle-Antti Suominen, and Artur Ekert. Quantum computers and dissipation. *Proc. R. Soc. Lond. A*, 452(1946):567–584, 1996. ISSN 1364-5021.

[5] John Preskill. Reliable quantum computers. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 454, pages 385–410. The Royal Society, 1998. ISBN 1364-5021.

[6] W. Daniel Hillis. What Is Massively Parallel Computing, and Why Is It Important?, 1992.

[7] Blaise Barney. Introduction to Parallel Computing, 2018. URL `https://computing.llnl.gov/tutorials/parallel{\_}comp/{\#}Whatis`.

[8] Blaise Barney. Message Passing Interface (MPI), 2019. URL `https://computing.llnl.gov/tutorials/mpi/`.

[9] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[10] Blaise Barney. OpenMP, 2018. URL `https://computing.llnl.gov/tutorials/openMP/`.

[11] Ewing Lusk and Anthony Chan. Early experiments with the OpenMP/MPI hybrid programming model. In *International Workshop on OpenMP*, pages 36–47. Springer, 2008.

[12] J Cheng, M Grossman, and T McKercher. *Professional CUDA C Programming*. EBL-Schweitzer. Wiley, 2014.

[13] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*, volume 21. Addison-Wesley, 2010.

[14] I C Kampolis, X S Trompoukis, V G Asouti, and K C Giannakoglou. CFD-based analysis and two-level aerodynamic optimization on graphics processing units. *Computer Methods in Applied Mechanics and Engineering*, 199(9):712–722, 2010. URL `http://www.sciencedirect.com/science/article/pii/S0045782509003648`.

[15] Everett Phillips, Yao Zhang, Roger Davis, and John Owens. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In *47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*, page 565, 2009.

[16] W Hwu Wen-Mei. *GPU computing gems emerald edition*. Elsevier, 2011. ISBN 0123849896.

[17] A. Frezzotti, G. P. Ghiroldi, and L. Gibelli. Solving model kinetic equations on GPUs. *Computers and Fluids*, 50(1):136–146, nov 2011.

[18] NVIDIA. GPU-Accelerated Ansys Fluent. URL `https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/ansys-fluent/`.

[19] MathWorks. MATLAB GPU Computing Support for NVIDIA CUDA Enabled GPUs - MATLAB & Simulink. URL https://www.mathworks.com/solutions/gpu-computing.html.

[20] Fengshun Lu, Junqiang Song, Xiaoqun Cao, and Xiaoqian Zhu. CPU/GPU computing for long-wave radiation physics on large GPU clusters. *Computers & Geosciences*, 41:47–55, apr 2012.

[21] Matthew Norman, Jeffrey Larkin, Aaron Vose, and Katherine Evans. A case study of CUDA FORTRAN and OpenACC for an atmospheric climate kernel. *Journal of Computational Science*, 9:1–6, 2015.

[22] Hamed Kargaran, Abdolhamid Minuchehr, and Ahmad Zolfaghari. The development of GPU-based parallel PRNG for Monte Carlo applications in CUDA Fortran. *AIP Advances*, 6(4):045101, apr 2016.

[23] Weylin MacCalla and Sameer Kulkarni. Utilizing GPUs to Accelerate Turbomachinery CFD Codes. 2016.

[24] Tobias Brandvik and Graham Pullan. Acceleration of a 3D Euler solver using commodity graphics hardware. In *46th AIAA aerospace sciences meeting and exhibit*, page 607, 2008.

[25] J D Anderson. *Hypersonic and High Temperature Gas Dynamics*. McGraw-Hill series in aeronautical and aerospace engineering. American Institute of Aeronautics and Astronautics, 2000.

[26] Zachary Chavis and Richard G Wilmoth. Plume Modeling and Application to Mars 2001 Odyssey Aerobraking. *Journal of Spacecraft and Rockets*, 42(3):450–456, may 2005. ISSN 0022-4650. doi: 10.2514/1.15171. URL https://doi.org/10.2514/1.15171.

[27] K Jousten and C B Nakhosteen. *Handbook of Vacuum Technology*. Wiley, 2016.

[28] H G Tompkins, American Vacuum Society. Education Committee, and T A Gessert. *The Fundamentals of Vacuum Technology*. American Vacuum Society monograph series. Education Committee, American Vacuum Society, 1997. ISBN 9781563967016. URL `https://books.google.gr/books?id=ZAlFAgAACAAJ`.

[29] M Gad-el Hak. *The MEMS Handbook*. Mechanical and Aerospace Engineering Series. Taylor & Francis, 2001. ISBN 9780849300776. URL `https://books.google.gr/books?id=No7znQEACAAJ`.

[30] Chih-Ming Ho and Yu-Chong Tai. Micro-electro-mechanical-systems (MEMS) and fluid flows. *Annual review of fluid mechanics*, 30(1):579–612, 1998.

[31] Daniel T Banuti, Martin Grabe, and Klaus Hannemann. Flow Characteristics of Micro-Scale Planar Nozzles. In *50th AIAA/ASME/SAE/ASEE Joint Propulsion Conference*, page 4002, 2014.

[32] Jeff C Taylor, Ann B Carlson, and H A Hassan. Monte Carlo simulation of radiating re-entry flows. *Journal of Thermophysics and Heat Transfer*, 8(3):478–485, jul 1994.

[33] S. Naris, C. Tantos, and D. Valougeorgis. Kinetic modeling of a tapered Holweck pump. *Vacuum*, 109:341–348, nov 2014.

[34] Ilkka Tittonen and Mika Koskenvuori. Electrostatic and RF-Properties of MEMS Structures. In *Handbook of Silicon Based MEMS Materials and Technologies (Second Edition)*, pages 294–312. Elsevier, 2015.

[35] F Sharipov. *Rarefied Gas Dynamics: Fundamentals for Research and Practice*. Wiley, 2016. ISBN 9783527413263.

[36] Martin Knudsen. Die Molekularströmung der Gase durch Offnungen und die Effusion. *Annalen der Physik*, 333(5):999–1016, jan 1909.

[37] D B Newell Zhang, F Cabiati, J Fischer, K Fujii, S G Karshenboim, H S Margolis, E de Mirandés, P J Mohr, F Nez, K Pachucki, T J Quinn, B N Taylor, M Wang, B M

Wood, and Z. The CODATA 2017 values of h , e , k , and N A for the revision of the SI. *Metrologia*, 55(1):L13, 2018. ISSN 0026-1394. URL http://stacks.iop.org/0026-1394/55/i=1/a=L13.

[38] C Cercignani and R Penrose. *Ludwig Boltzmann: The Man Who Trusted Atoms*. OUP Oxford, 2006.

[39] P L Bhatnagar, E P Gross, and M Krook. A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems. *Physical Review*, 94(3):511–525, may 1954.

[40] V G Leitsina and N V Pavlyukevich. On the temperature jump in a rarefied gas over a permeable surface. *Journal of engineering physics*, 19(2):974–978, 1970. ISSN 1573-871X. doi: 10.1007/BF00828771. URL https://doi.org/10.1007/BF00828771.

[41] E M Shakhov. Generalization of the Krook kinetic relaxation equation. *Fluid Dynamics*, 3(5):95–96, 1968.

[42] Lowell H Holway Jr. New statistical models for kinetic theory: methods of construction. *The physics of fluids*, 9(9):1658–1673, 1966.

[43] A. B. Huang and D. L. Hartley. Nonlinear Rarefied Couette Flow with Heat Transfer. *Physics of Fluids*, 11(6):1321, aug 1968. ISSN 00319171. doi: 10.1063/1.1692103. URL https://aip.scitation.org/doi/10.1063/1.1692103.

[44] G A Bird. *The DSMC Method*. CreateSpace Independent Publishing Platform, 2013. ISBN 9781492112907.

[45] Felix Sharipov and Vladimir Seleznev. Data on Internal Rarefied Gas Flows. *Journal of Physical and Chemical Reference Data*, 27(3):657–706, may 1998. ISSN 0047-2689. doi: 10.1063/1.556019. URL https://doi.org/10.1063/1.556019.

[46] The Portland Group. CUDA Fortran Programming Guide. Technical report, 2018.

[47] Massimiliano Fatica and Gregory Ruetsch. *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming*. 2013.

[48] National Research Council (U.S.). Committee on the Fundamentals of Computer Science: Challenges and Opportunities. *Computer science : reflections on the field, reflections from the field*. National Academies Press, 2004.

[49] Mark Harris. Write Flexible Kernels with Grid-Stride Loops, 2013.

[50] Simon Green. Particle simulation using cuda. *NVIDIA whitepaper*, 6:121–128, 2010.

[51] John Rethans. APIs: Leverage For Digital Transformation, 2016. URL `https://www.forbes.com/sites/forbestechcouncil/2017/05/08/apis-leverage-for-digital-transformation/{\#}4895c1577140`.

[52] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, page 483, New York, New York, USA, 1967. ACM Press.

[53] John D C Little and Stephen C Graves. Little's Law. In *Chhajed D., Lowe T.J. (eds) Building Intuition. International Series in Operations Research & Management Science*, pages 81–100. Springer, 2008.

[54] Vasily Volkov. Better Performance at Lower Occupancy. Technical report, UC Berkeley, 2010.

[55] NVIDIA Docs. Achieved Occupancy. URL `https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm`.

[56] M. Harris. How to Implement Performance Metrics in CUDA C/C++ | NVIDIA Developer Blog, 2012. URL `https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/`.

[57] TechPowerUp. NVIDIA Quadro M2200 Mobile Specs | TechPowerUp GPU Database, 2017. URL https://www.techpowerup.com/gpu-specs/quadro-m2200-mobile.c2922.

[58] TechPowerUp. NVIDIA TITAN Xp Specs | TechPowerUp GPU Database, 2018. URL https://www.techpowerup.com/gpu-specs/titan-xp.c2948.

[59] Jeff Larkin. Chapter 2 - Profile-guided development with OpenACC. pages 29–53. Morgan Kaufmann, Boston, 2017.

[60] Y Cengel. *Heat And Mass Transfer: Fundamentals And Applications*. 2014.

[61] ASM International. Handbook Committee. *Properties and Selection: Nonferrous Alloys and Special- Purpose Materials*. Metals handbook. ASM International, 1990.

[62] N A Nurlybaev. Discrete velocity method in the theory of kinetic equations. *Transport Theory and Statistical Physics*, 22(1):109–119, feb 1993.

[63] S Chapra. *Numerical Methods for Engineers*. 2014.

[64] Paraskevi Ladopoulou. Parallel Programming of Scientific Applications Using GPU/CUDA. 2014.

[65] Sarantis Pantazis. *Simulation of transport phenomena in conditions far from thermodynamic equilibrium via kinetic theory with applications in vacuum technology and MEMS*. PhD thesis, University of Thessaly, 2011.

[66] Christos Tantos. *Effect of rotational and vibrational degrees of freedom in polyatomic gas heat transfer, flow and adsorption processes far from local equilibrium*. PhD thesis, University of Thessaly, 2016.

[67] Mark Harris. Unified Memory for CUDA Beginners, 2017. URL https://devblogs.nvidia.com/unified-memory-cuda-beginners/.

[68] Dana Jacobsen, Julien Thibault, and Inanc Senocak. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, page 522, 2010.

[69] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, 182(1):266–269, 2011. ISSN 0010-4655.

[70] Harris Mark. Optimizing parallel reduction in CUDA. *NVIDIA CUDA SDK*, 2, 2008.