Optimisation and development of an eddy-viscosity turbulence model for a GPU Enabled, Compressible Flow Solver

Jean-Francois WALD — 40092604
Master - Final Year Project - Queen’s University Belfast
School of Mechanical & Aerospace Engineering

Supervisor :
Dr. Simao Marques

September 28, 2012
Abstract

The use of Graphical Precessing Units (GPU) to do general purpose computations is a quite new and promising field of research. Indeed, boosted by the video game industry, GPU acquired a massive and powerful parallel architecture. The advent of modern graphics cards and programming tools offer the possibility of a significant speed up of CFD computations over price-equivalent, conventional CPU architectures. However it is use in CFD is hampered by the memory requirements of typical CFD problems. Low memory CFD solvers can be developed by using explicit time integration methods. This study focused on the development of an eddy-viscosity model for a GPU Enabled, Compressible Flow Solver as well as on the optimization of the code in order to take fully advantage of the GPUs capabilities. Due to its low memory requirements, this study will focused on Spalart-Allmaras model and provide some initial results for further improvement. Some advances towards a domain decomposition strategy ported to GPU is also proposed, in order to obtain a fast and powerful RANS solver.

Resume

L’utilisation d’Unites de Calcul Graphique (GPU) a des fin autres que l’affichage graphique est un secteur de recherche prometteur et relativement recent. En effet, pousse par l’industrie du jeu video, les GPU ont acquis une architecture massivement parallele, depassant de loin la puissance theorique de la plupart des stations de travail. Les recentes avancees en matiere de cartes graphiques ainsi que d’outils de programmation offrent la possibilite d’otenir des accelerations signifcatives lors de calcul CFD par rapport aux architecture CPU classiques. Cependant, l’utilisation de GPU en CFD est encore limite, notamment par la gestion complexe de la memoire qu’exige un GPU. Cette etude s’articule autour de deux axes principaux. Dans un premiers temps une tentative de developpement d’un modele a viscosite turbulente dans un code compressible et accelere par GPU a ete effectue. Dans un second temps, cette etude se concentre sur l’optimisation de la version Euler du code. Une approche novelle a decomposition de domaine est envisagee, et permet d’obtenir des accelerations significative de l’ordre de 15.
Acknowledgment

Firstly, and most importantly, I would like to deeply thank my supervisor, Simao Marques, whose help, advice and supervision was invaluable.

I also wish to express my sincere thanks to Steven Allmaras and Philippe Spalart for the time they took to answer my questions.

I would also like to thank Erwan Collin whose disposibility and quick help was precious.

Finally, I also place on record my sense of gratitude to one and all who, directly or indirectly, have made this project possible or have lent their helping hand in this project.
Contents

I  Introduction  10

1  Parallel computing : MPI, OpenMP  10

2  CPUs vs GPUs  12

3  GPU programming languages  12

4  Turbulence simulation  14

5  Aims and objectives  15

II  Literature review  16

1  Computational Fluid Dynamics  16
   1.1  Discretization techniques  16
   1.2  Numerical schemes  17
      1.2.1  Upwind Difference Scheme (UDS)  17
      1.2.2  Central Difference Scheme (CDS)  17
      1.2.3  Higher-order scheme  18
   1.3  Turbulence modeling  18
      1.3.1  Direct Numerical Simulation (DNS)  19
      1.3.2  Large Eddy Simulation (LES)  19
      1.3.3  Reynolds Average Navier-Stokes (RANS)  20

2  GPU computing  23

III  Theory and numerical methods  26

1  Compressible Navier-Stoke equations  26
   1.1  Conservation equations  26
   1.2  Comportements law  27
   1.3  States law  27

2  Averaging operators  28
   2.1  Reynolds averaging  28
2.2 Favre’s Averaging ......................................................... 28

3 Averaged Navier-Stokes Equations ........................................ 29
  3.1 Continuity equation ..................................................... 30
  3.2 Momentum conservation equation .................................... 30
  3.3 Energy conservation equation ....................................... 31

4 Boussinesq Hypothesis ..................................................... 33

5 Spalart-Allmaras model .................................................... 34
  5.1 Motivations .............................................................. 34
  5.2 General incompressible formulation .................................. 35
    5.2.1 Diffusion term ....................................................... 35
    5.2.2 Production term ..................................................... 36
    5.2.3 Destruction term .................................................... 36
    5.2.4 Calibration of the model ......................................... 36
    5.2.5 Final formulation .................................................. 38
  5.3 Compressible Spalart-Allmaras model ............................... 40

6 Numerical methods ........................................................ 41
  6.1 Problem description ................................................... 41
  6.2 Finite volume formulation ............................................ 42
    6.2.1 Space discretization ............................................... 43
    6.2.2 Time integration .................................................... 44
  6.3 Artificial dissipation .................................................. 45

IV GPU implementation ...................................................... 47

1 Hardware ................................................................. 47

2 GPU program structure ................................................... 48

3 Grids, Blocks and Threads ............................................... 50

4 Example on a Matrix/Vector multiplication ............................ 51

5 Application to a RANS Navier Stokes solver ........................... 54
  5.1 Strategy ............................................................... 54
  5.2 Code architecture .................................................... 56
V Preliminary results

VI Performance and optimisation

1 Brut performance

2 Level of optimisation
   2.1 Optimisation of algorithms for the GPU
   2.2 Optimization of Memory Accesses
   2.3 The advantage of shared and local memory
   2.4 Optimization of the parallel environment set up
   2.5 Avoiding divergent branches

3 Optimized performance

4 Domain decomposition based algorithm
   4.1 Initial conjecture
   4.2 1D compressible Euler equation
   4.3 2D compressible Euler equations
      4.3.1 Metis software
      4.3.2 Reordering
      4.3.3 Halo cells
      4.3.4 Padding
      4.3.5 Thread cooperation
      4.3.6 Implementation and validation
      4.3.7 Performance

VII Conclusion

VIII Annex
List of Figures

1. Moore’s Law ................................................................. 10
2. An IBM Blue Gene supercomputer ........................................ 11
3. Evolution of CPU and GPU theoretical power since 2001 .......... 12
4. Global architecture of CPUs and GPUs ................................. 13
5. Classical representation of the energy spectrum in isotropic homogeneous turbulence ......................................................... 19
6. Schematic representation of turbulent motion (left) and time dependence of a velocity component (right) (taken from J. H. Ferziger’s book [12]) ................................. 20
7. Velocity profile of a turbulent boundary layer. Theoretical linear and logarithmic profiles and DNS data. .................................................. 37
8. Control volume Ω ............................................................. 42
9. Control volume with N edges ................................................ 43
10. Control volume Ω ............................................................. 43
11. Auxiliary control volume defined to compute gradients at cell faces .................................................. 44
12. CPUs and GPUs have fundamentally different design philosophies .................................................. 47
13. Basical execution of a CUDA program ........................................ 49
14. Layout of Grids, Blocks and Threads ........................................ 50
15. Matrix/Vector time comparison .............................................. 54
16. .................................................................................. 54
17. An illustration of a race condition when running code on parallel environment. On the top, a serial CPU execution: the memory location for the variable of cell a is sequentially accessed. On the bottom, a parallel execution: memory accessing by three threads at the same time. cores at the same time .................................................. 55
18. Example of a grid before and after the colouring process. All three edges of one cells are mandatorily of different colours, so that if colours are sequentially computed, two edges cannot access the same cell. .................................................. 56
19. Sequential execution of each colour ........................................... 56
20. Global structure of the code ................................................... 57
21. RAE2822 grid for RANS calculation ........................................ 58
22. Convergence history for test case ............................................. 59
23. Pressure coefficient comparison ............................................... 60
24. Pressure, Mach, Velocity and turbulent viscosity contours .......... 61
25. Kernelized time-stepping and dissipation subroutine time execution .................................................. 62
26. Illustration of coalescent and non coalescent memory access in the GPU .................................................. 63
Illustration of non coalescent memory access for edge-based strategy on the GPU.

Illustration of loading data in shared memory. The first access is highly non coalescent, but then all accesses of the shared memory will be faster and coalescent.

Execution time of 10 matrix/vector multiplication, $N = 13440$.

3 types of grids used to compare results.

Time comparison on coarse grid.

Time comparison on medium grid.

Time comparison on fine grid.

Difference of number of memory access between a cell based (CB) and an edge based (EB) algorithms, with a JST scheme.

Analytical and numerical solution at $t=0.2s$ for the Sod shock tube problem.

28 subdomain on a NACA0012 profile comporting 10682 cells.

Illustration of memory location of data relative to each cell before/after reordering process.

Illustration of halo cells principle, and their contiguous locations in memory.

Illustration of padding principle. A padding of 18 is illustrated here: a block of contiguous data in memory contains the cells + halo cells + cells need to fill space up to 18 (padding cells).

Naca0012 grid.

Convergence.

RAE2822 grid.

Naca0012 grid.

Convergence.

Pressure coefficient from our code and results from Jameson.

Naca0012 grid.

Time comparison between the serial, parallel edge based and domain decomposition code.
Nomenclature

\( \delta_{ij} \) Kroenecker symbol

\( \gamma \) Laplace perfect gas constant

\( \kappa \) thermal conductivity

\( \lambda \) dynamic viscosity

\( \gamma = (\gamma_1, \gamma_2)^T \) heat flux vector

\( \mathbf{u} = (u_1, u_2)^T \) velocity vector

\( \mathbf{x} = (x_1, x_2)^T \) position vector

\( \mu \) volume viscosity

\( \mu_T \) Turbulent eddy viscosity

\( \nu \) kinematic viscosity

\( \nu_t \) kinematic turbulent viscosity

\( \rho \) density

\( \sigma_{ij} \) stress tensor

\( c_p \) thermal capacity at constant pressure

\( c_v \) thermal capacity at constant volume

\( E \) total energy

\( e \) internal energy

\( k \) turbulent kinetic energy

\( p \) pressure

\( Pr \) Prandlt number

\( Pr_T \) turbulent Prandlt number

\( r \) perfect gas universal constant

\( R_{ij} \) Reynolds stress tensor
\( s_{ij} \) strain tensor

\( T \) temperature

\( t \) time

**Operators and index**

\( \overline{\cdot} \) Reynolds averaging operator

\( \tilde{\cdot} \) Favre averaging operator

\( \cdot' \) Fluctuating part in Reynolds averaging operator

\( \cdot'' \) Fluctuating part in Favre averaging operator

**Abreviation**

GPGPU General Purpose Graphical Processing Unit

GPU Graphical Processing Unit

CPU Central Processing Unit

VRAM Video Random-access Memory

EB Edge-base

CB Cell-base

RANS Reynolds Average Navier-Stokes

DNS Direct Numerical Simulation

LES Large Eddy Simulation

CUDA Compute Unified Device Architecture

SM Streaming Multiprocessor

SP Streaming Processor
Part I

Introduction

Computational fluid dynamics constitutes the “third approach” in the study and development of the whole discipline of fluid dynamics. In the seventeenth century, the foundations for experimental fluid dynamics were laid in France and England. Then the eighteenth and nineteenth century saw the gradual birth of theoretical fluid dynamics, again in Europe. The advent of computer sciences in the 70’s made computational fluid dynamics possible, which greatly helped researchers to understand and interpret the results of theory and experience, and vice versa. But the need to simulate always bigger problems (aircraft industry, cars industry...) bring the need to find solution, on the one hand to run faster simulation, through High Performance Computing (HPC), on the other to model the small scales of a fluid motion, which are highly time consuming.

1 Parallel computing : MPI, OpenMP

The computational power of Central Processing Units (CPUs) has been growing at a steady rate over the past few decades, the number of transistors in CPUs roughly doubling in power every two years in accordance to Moores Law (Figure 1).

![Figure 1: Moore’s Law](image)
This allowed the construction of increasingly faster supercomputers, allowing the computation of larger and larger mathematical problems. This technology has also provided desktop and laptop computers with computational power that a decade before would have classified as a supercomputer in itself. What tends to go unnoticed is that although modern CPUs run at very high clock speeds, their serial architecture hinders their potential mathematical capability. All CPUs perform operations sequentially - albeit at tremendous speed - allowing them to deal with calculations quickly. Parallel computation has recently become necessary to take full advantage of the gains allowed by Moore’s law. High Performance Computing (HPC) improved this further by running several processors in parallel, allowing operations to be done simultaneously, as opposed to sequentially.

Some supercomputers reached the phenomenal number of 40,000 cores (Figure 2). But the record is detained by Japan with its K-Computer (705,024 cores).

Figure 2: An IBM Blue Gene supercomputer

Many parallel programming languages and models have been proposed in the past several decades. The ones that are the most widely used are the Message Passing Interface (MPI) for scalable cluster computing and OpenMP for shared-memory multiprocessor systems. MPI is a model where computing nodes in a cluster do not share memory; all data sharing and interaction must be done through explicit message passing. MPI has been successful in the high-performance scientific computing domain. Applications written in MPI have been known to run successfully on cluster computing systems with more than 100,000 nodes. However, performance is limited by the communication network between the nodes. The amount of effort required to port an application into MPI can be extremely high. OpenMP supports shared memory, however it has not been able to scale beyond a couple hundred computing nodes due to thread management overheads and cache coherence hardware requirements. [20]
2 CPUs vs GPUs

Since 2003, the semiconductor industry has settled on two main trajectories for designing microprocessors. The multicore trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. This laid to the birth of the recent Intel Core i7 microprocessor, which possess 4 cores designed to maximize the execution speed of sequential programs.

In contrast, the many-core trajectory focuses more on the execution of parallel applications. The many-cores began as a large number of much smaller cores, and, once again, the number of cores doubles with each generation.

This many-core philosophy is typical of Graphical Processing Units (GPUs). Indeed, modern GPUs contain hundreds of cores all operating in parallel, similar to HPC systems. This means that they have the potential for huge computational output if used correctly. A single modern graphics card can exceed a TFLOP, whereas even the most recent CPU can only attain several hundred GFLOPs (Figure 3).

3 GPU programming languages

The design philosophy of the GPUs is shaped by the fast growing video game industry, which exerts tremendous economic pressure for the ability to perform a massive number of floating-point calculations per video frame in advanced games. This demand motivates the GPU vendors to look for ways to maximize the chip area and power budget dedicated to floating-point calculations. The competition between hardware manufacturers AMD and NVidia has
yielded not only highly powerful graphics cards, but they have also remained at an affordable price, considering their potential computational output.

GPUs also have the advantage over CPUs in that they use far less electrical power for the same computational output. CPUs require large amounts of electrical power to maintain their high clock speeds, and although GPU cores run at a lower clock speed, there are far more of them. However, the architecture of GPUs means this electrical power is converted to computational power far more efficiently, meaning that GPUs provide far more FLOPs per unit power than CPUs. This was illustrated in 2009 when server maker Novatte produced a cluster that combined 60 Intel Xeon CPUs and 120 Nvidia Tesla GPUs. The result required 28% less power to run than a cluster with similar computational output that uses Xeon CPUs exclusively. As of November 2011, three of the top five supercomputers in the world adopt GPUs as accelerators [1]. The upcoming Titan supercomputer at Oak Ridge National Laboratory (expected to make its debut in early 2013) is projected to be the fastest supercomputer in the world with a peak performance of over 10 petaflops (i.e. 10 trillion floating point operations per second). Titan will be powered by NVIDIA GPUs that is expected to power efficient. Furthermore, an exascale supercomputer is expected to arrive in 2018.

Until 2006, graphics chips were very difficult to use because programmers had to use the equivalent of graphic application programming interface (API) functions to access the processor cores, meaning that OpenGL or Direct3D techniques were needed to program these chips. This technique was called GPGPU, short for general-purpose programming using a graphics processing unit. But since GPU computing begin to sparks the interest of the scientific community, GPU manufacturer have developed well-suited application programming interface.

![CPU/GPU Architecture Comparison](image)

Figure 4: Global architecture of CPUs and GPUs
This is why the CUDA (Compute Unified Device Architecture) programming model, introduced by NVidia in 2007, is designed to support joint CPU/GPU execution of an application. CUDA essentially gives software developers simple access to the GPU through a runtime library, which is now available in all major languages such as C, Fortran, Java and Python. Utilizing this, developers can create software that interfaces with the GPU and thus take advantage of their massively parallel environment to any number of applications, such as CFD.

Since its release, CUDA has attracted a huge user base from a wide variety of fields, meaning there are numerous sources of information and help regarding its use. Consequences are that this report will utilize CUDA in a C/C++ environment.

4 Turbulence simulation

One must not forget the physical aspect of every simulation. As fast as the (super)computer can be, it is far not sufficient to obtain good numerical results. Using 100% of the largest supercomputer (K-computer) would only allow the simulation of one Direct Numerical Simulation of a car at 3 km/h [24]! The purpose of DNS is to simulate all the scales of turbulence, from the smallest to the biggest eddies, directly from the Navier-Stokes equations. But the the difference in scales are sometimes so big that it is even today limited to some research studies. Hence the need to introduce models that will allow to simulate only some scales of a turbulent motion.

It exists two main approach to model the turbulence. The first one, Large Eddy Simulation (LES), consists in simulating only the biggest eddies while modeling the smallest ones. But, due principally to the treatment of the walls, the computational cost of this technique is still relatively high. More suitable for an industrial context the second one is called the Reynolds Average Navier-Stokes (RANS) turbulence modeling technique.

Indeed, engineers are normally interested in knowing just a few quantitative properties of a turbulent flow, such as the average force on a body for example. Contrary to LES approach, RANS strategy is to model all the scales of turbulence (ie. all Kolmogorov’s spectrum). Inherited from Osborne Reynolds works, RANS approach has always been the most popular method in industries, due to its little computational cost. Turbulence is considered like a completely stochastic process in which only statistics are supposed previsible. Thus, there is no need to have extremely fine grids. However, the greatest weakness of this method is its empiricism. Indeed, each model applies to a very particular type of flow. However, it appears to be the better solution to take fully advantage of the power of GPUs to solve the complete Navier-Stokes equation from an industrial point of view.
5 Aims and objectives

This project is articulated around three main objectives:

- **Port a CFD code to GPU**: the current code is partially ported to GPU. First objective is to get a fully working Euler GPU version of the code. The objective is then to perform some calculations in order to get some results. Performance tests can then be made in order to identify bottlenecks for later optimization.

- **Full exploitation of GPU parallelism and power**: in order to take fully advantage of the GPU, the code need optimization. First aim will thus be to look works that have been done before in order to identify the different way of optimization. Again, performance tests can then be run to compare to previous results.

- **Develop a RANS version of the code**: as developed in section 4 industrial computation deals with real viscous fluids at high Reynolds number. Hence the need to develop a RANS version of the code that take fully advantage of the GPU capability. A validation according to experiments will then be necessary.
Part II

Literature review

1 Computational Fluid Dynamics

1.1 Discretization techniques

The *Finite Difference Method*, or FDM, which is based on the properties of Taylor expansions, is probably the simplest method to apply and implement. However, this differencing technique requires calculations on a structured grid, which makes the use of complex geometries difficult. Nevertheless, a fictitious domain approach [26, 25] has recently been developed for Computational Fluid Dynamic by Khadra et al. [19] and allows the use of complex geometries.

The FDM uses the strong or differential form of the governing equations. Another type of discretization rely on the integral form of the Navier-Stokes equations: finite element and finite volume methods. Indeed, classical FDM can be expected to break down near discontinuities in the solution where the differential equation does not hold. Rather than pointwise approximations at grid points, the *Finite Volume Method* (FVM) break the domain into grid cells and approximate the total integral of a quantity $q$ over each grid cell, or actually the cell average of $q$, which is this integral divided by the volume of the cell. These values are modified in each time step by the flux through the edges of the grid cells. It has apparently been introduced in the field of CFD by McCormack and Paudlay (1972) and is particularly well suited for the numerical simulation of various types. It has been extensively used in several engineering fields such as fluid mechanics and may be used on arbitrary geometries, using structured or unstructured meshes. The local conservativity of the numerical fluxes is one of the most important property of the method: the numerical flux is conserved from one discretization cell to its neighbor. This makes the Finite Volume Method closer to the physics and quite attractive for fluid mechanics.

Borrowed from Structural Analysis, the *Finite Element Method* (FEM) is another numerical technique for finding approximate solutions of partial differential equations. It was first developed in 1943 by R. Courant. The most attractive property of the FEM is its ability to handle complicated geometries (and boundaries) with relative ease. It is however not widely used in CFD, and Finite Volume is generally preferred.
1.2 Numerical schemes

The approximations to the integrals requires the values of variables at locations other than computational nodes. Indeed, the most relevant problem when using FVM is to determine good numerical flux functions that approximate the correct fluxes reasonably well, based on the approximate cell averages, the only data available. A lot of those schemes have been developed, each of them having their drawbacks and advantages.

1.2.1 Upwind Difference Scheme (UDS)

This interpolation simply consists in approximating the fluxes by its value at the node upstream. This is the only scheme that unconditionally satisfies the boundedness criterion, i.e. it will never yield oscillatory results. However, it achieves this by being numerically very diffusive. With its first order accuracy in space, it introduces much more diffusion than is actually required, and gives numerical results that are typically badly smeared unless a very fine grid is used.

1.2.2 Central Difference Scheme (CDS)

Another straightforward way to express the fluxes through the edges of the grid cells is probably to use linear interpolation between the two nearest nodes. A first attempt might be the simple arithmetic average of the neighbor cells. Unfortunately, this method is generally unstable for hyperbolic problems and cannot be used, even if the time step is small enough that the CFL condition is satisfied.

Lax-Friedrichs (1954) scheme present the advantage to be centered and very stable. However, like UDS schemes, it presents the drawback to be too diffusive. The first centered scheme to be introduced, with second order accuracy in space and time levels, is due to Lax-Wendroff (1960). It has been the most widely applied scheme for aeronautical applications, up to the end of 1980s under various form (Richtmyer scheme, McCormack scheme, see [14, 21]). But it actually presents a lot of oscillations near discontinuities like shocks and requires sometimes a lot of computer memory, which is not compatible with an acceleration via GPUs.

Jameson, Schmidt and Turkel had the idea to simply add an artificial dissipation term to classical center scheme that shown himself unconditionally unstable for Euler’s equations. The purpose of Jameson, Schmidt and Turkel was to “develop economical methods of solving the Euler equations, particularly for steady flows, with the aim of reducing the computational cost to the point where they might be used as an alternative to potential flow calculations for design work.” [17]. This low memory requirement combined with its very robust behavior make this scheme the ideal candidate for an accurate finite volume code accelerated by GPU.
1.2.3 Higher-order scheme

- **Flux Vector Splitting (FVS)**
  Center schemes don’t respect the sense of propagation of the information, hence the necessity for some applications of numerical expression of fluxes that take into account this sense of propagation. Fluxes across each edges are decomposed into components that flow along the various characteristics of the equations. Steger and Warming and then Van Leer in 1982 introduced first order schemes based on Flux Vector Splitting method. The resulting method is fairly complex and is sometimes considered as too much diffusive for the complete Navier-Stokes equations. However, those schemes are robust and allow the calculation of strong shocks.

- **Based on the resolution of Riemann’s problem**
  Still more sophisticated methods (Gudunov scheme for example) consists in approaching the solution by a piecewise constant function. Then at every cell interface, the corresponding Riemann problem is solved. This method have a lot of physical advantages, as it satisfies for example the entropy condition, but actually requires a prohibitive computational cost. Roe’s idea was to replace this exact Riemann problem by an approximate linear problem. All those schemes are commonly used today, and are known as FDS schemes (Flux Difference Splitting).

- **TVD second order extension, MUSCL and limiters**
  To allow FDS and FVS schemes to be second order accurate, a natural method consists in approaching the solution by a piecewise linear, and no more constant, functions. This is called MUSCL reconstruction (Monotonic Upstream-Centered scheme for Conservation law). To avoid oscillations, Harten introduces in 1983 the concept of Total Variation Diminishing (TVD) schemes that uses some functions called limiters.

  FDS and FVS schemes are considered as high resolution numerical schemes. They require most of the time a lot of memory, and are really complicated to implement. Thus, they are not particularly adapted to our needs. One of the more accurate and robust scheme, developed especially for low memory treatment remains the JST scheme.

1.3 Turbulence modeling

A turbulent flow is characterized by a hierarchy of scales through which an energy cascade takes place. Dissipation of kinetic energy takes place at scales of the order of Kolmogorov length $\eta$ (length of the smallest eddies), while the input of energy into the cascade comes
from the decay of the large scales, of order $L$.

By dimensional analysis Kolmogorov proposed to express those scales by:

$$\eta = \left( \frac{\nu^3}{\varepsilon} \right)^{1/4} \quad L = \left( \frac{k^{3/2}}{\varepsilon} \right)$$  \hspace{1cm} (1)

where $\varepsilon$ is the energy dissipation rate, $k$ the turbulent kinetic energy and $\nu$ the viscosity of the fluid. These two scales at the extremes of the cascade can differ by several orders of magnitude at high Reynolds numbers. In between there is a range of scales (each one with its own characteristic length $r$) called inertial zone in which energy is transferred from the biggest to the smallest scales. The spectral density of energy is generally represented as figure 5.

![Figure 5: Classical representation of the energy spectrum in isotropic homogeneous turbulence](image)

### 1.3.1 Direct Numerical Simulation (DNS)

Direct Numerical Simulation "simply" consists in solving the complete Navier-Stokes equations. All the scales of turbulence are solved, no models are required. It is however not used in industry, due to its prohibitive computational cost. It can for example be shown that the cost of a DNS on a car at 100 km/h is of about $10^{27}$ FLOPs. This is far from the theoretical power of the biggest supercomputers. Actually, an extrapolation of Moore’s law show that DNS will only be possible in car or aerospace engineering in 2080 (evaluation from Spalart, Boeing [29], table 1). Obtaining the complete range of turbulent scales is not possible hence the necessity of modeling approaches where the small scales are not resolved, which makes turbulence modeling one of the key elements in Computational Fluid Dynamics.

### 1.3.2 Large Eddy Simulation (LES)

As we noted, turbulent flows contain a wide range of length scale. The principle of LES is very simple : it consists in explicitly compute the large-scale motion while modeling the smaller
scales as illustrated in figure 6.

![Figure 6: Schematic representation of turbulent motion (left) and time dependence of a velocity component (right)](taken from J. H. Ferziger’s book [12])

But CPU cost for LES remains very important. The situation is even more difficult for wall bounded flows (close to the wall, the energetic turbulent structures are at very small scale). So although this type of modelisation is really close to physics and requires few empiricism, it is not well-fitted for the moment for industrial application (see table 1).

### 1.3.3 Reynolds Average Navier-Stokes (RANS)

The following table gives the reader an overview of the strength and the weakness of those different method. DNS and LES methods are, from a physical and mathematical point of view, much more satisfying, but are not ready to enter the industry environment yet. Hence the importance from an industrial point of view to implement and develop RANS turbulence models.

<table>
<thead>
<tr>
<th>Name</th>
<th>Re-Dependence</th>
<th>Empiricism</th>
<th>Grid cells</th>
<th>Time steps</th>
<th>Ready</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANS</td>
<td>Weak</td>
<td>Strong</td>
<td>$10^7$</td>
<td>$10^3$</td>
<td>1985</td>
</tr>
<tr>
<td>LES</td>
<td>Weak</td>
<td>Weak</td>
<td>$10^{11.5}$</td>
<td>$10^{6.7}$</td>
<td>2045</td>
</tr>
<tr>
<td>DNS</td>
<td>Strong</td>
<td>None</td>
<td>$10^{16}$</td>
<td>$10^{7.7}$</td>
<td>2080</td>
</tr>
</tbody>
</table>

Table 1: Summary of strategies according to Spalart [29] for external flows applications

### Algebraic model

The concept of RANS turbulence modeling came far before the birth of computers. Some pioneers like Joseph Valentin Boussinesq in 1877 introduced the concept of eddy viscosity, which describes the global effect of turbulence as diffusive. Boussinesq proposed relating the turbulence stresses to the mean flow to close the system of equations via this eddy viscosity.
It allows to replace the Reynolds stress tensor \( u_i u_j \), that translate the influence of the small scales on the biggest, by a model in the Navier-Stokes equations. It introduces the turbulent viscosity \( \nu_t \).

However, this **algebraic model** or **zero-equation-model**, does not provide a relation to evaluate \( \nu_t \) and \( k \), which may vary and have to be adapted or estimated depending on the case considered. In 1925, Prandtl introduces the concept of mixing length \( l_m \) which enables the evaluation of \( \nu_t \) via the equation: \( \nu_t = l_m^2 \sqrt{2 S_{ij} S_{ij}} \) where \( S_{ij} \) is the strain-rate tensor. This model can be used for very simple cases. Physically, the mixing length “may be considered as the diameter of the masses of fluid moving as a whole in each individual case; or again, as the distance traversed by a mass of this type before it becomes blended in with neighbouring masses” (Prandtl, 1926).

**One equation model**

It was Kolmogorov who introduced the concept of the well-known RANS modeling. Indeed, Prandtl and Kolmogorov proposed to compute \( k \) using a transport equation and then link \( \nu_t \) and \( k \) by a relation: \( \nu_t = C_\mu \sqrt{k l} \). This is the first **one-equation model**, \( C_\mu \) is a constant and \( l \) need to be specified, depending on the flow conditions. However those models doesn’t satisfy the completeness condition, which states that a model do not have to require a priori knowledge to be run.

The only one-equation model used in industrial applications is the Spalart-Allmaras model (1992). The model used a transport equation for \( \nu_t \) and is popular in aeronautics. It is really easy to integrate numerically, produces results as good as zero equations model for attached flows and gives a much better description of the flow field than zero equation models. However, the model is too simple (only one equation) to be valid in a wide range of flows.

**Two equation model**

A dimensional analysis can lead to the following expression of the turbulent viscosity:

\[
\nu_t = C_\mu \frac{k^2}{\varepsilon}
\]  

One of the most obvious choice to calculate this turbulent viscosity is then to calculate \( k \) and \( \varepsilon \). This is why Jones and Launder, using the previous work of Davidov (1961), Harlow and Nakayama (1967) and Hanjali (1970), proposed and applied a **two-equation model** using \( \varepsilon \) and \( k \). The final form of the model will be given by Launder and Spalding (1974). This is the standard \( k - \varepsilon \) model, by far the most widely used model in the industry.
Some possibilities to improve the behavior of the standard \( k-\varepsilon \) model were then proposed. Durbin (1996) proposed a correction that delete the stagnation point anomaly, which lead in an over production of turbulent energy near the border of attack of an airfoil for example. He remarked that this anomaly can be related to a realizability problem. Shih et al. (1995), from a similar reasoning on the realizability of the Reynolds stress, propose a variable \( C_\mu \) coefficient. This model is most of the times proposed by commercial codes, under the name Realizable \( k-\varepsilon \) model. By applying to turbulence the theory of the renormalization group, coming from theoretical physics, one can recover the equations of the \( k-\varepsilon \) model. Yakhot et al. (1992) deduced from this theory a RNG \( k-\varepsilon \) model.

Coming back to Kolmogorov's idea of using the characteristic frequency of the large eddies \( \omega = \varepsilon/k \) as the second scale, Wilcox (1988) builds a transport equation for \( \omega \) in the same form as the \( \varepsilon \) equation, and uses a linear constitutive relation. In external aerodynamics (e.g., a wing), the model predicts a boundary layer development sensitive to the turbulence level present outside of the boundary layer, which is usually fixed arbitrarily by the user. Thus, this model is not applicable to external aerodynamics. However, it is used as a near-wall model associated to the \( k-\varepsilon \) model far from the wall (\( k-\omega \) SST model, developed Menter, 1993). The idea of Menter was then to write a set of equation that tends to the \( k-\omega \) model in the near-wall region and to the \( k-\varepsilon \) model far from the wall. The transition between the two models is made by quite complex functions which depend on the distance to the wall. A lot of the development that are done today concerns this transition.

All those models are known as Eddy-Viscosity Models. They contain less differential equations than Reynolds stress models. This means that they are less expensive in terms of CPU time, although the difference is not always large. They are not necessarily more numerically stable than Reynolds stress models as stability depends on the linearity of the model. However, Eddy-viscosity models assumes a relation between the Reynolds stress and the mean field which this is not true in general. It is assumed that the response to a change in the strain is instantaneous which this is wrong in general. Actually, turbulence is sensitive to the history of the strain (memory effect).

**Reynolds stress models**

Going to Reynolds stress models (second moment closures) have decisive advantages. Indeed, since the transport equations of the Reynolds stresses \( u_i\bar{u}_j \) are solved, the models reproduce the memory effect of turbulence: for instance, in the case of a sudden change in the mean
strain, turbulence respond with a time delay. The Reynolds stress modeling thus represent a major breakthrough compared to eddy-viscosity models.

However, they most of the time require much more memory, which is not well-suited for GPU applications. For further details, one can refer to [?] .

2 GPU computing

Although porting CFD codes to a graphics card is a relatively new field of research, utilizing GPUs for scientific purposes is not. General Purpose computation on a GPU (GPGPU) was coined by Mark Harris [13] in his PhD Real-Time Cloud Simulation and Rendering in 2003, where he proposes the use of GPUs in simulating Rayleigh scattering.

Since then, GPGPU has been adopted in numerous different fields, from astrophysics to finance, all taking advantage of the raw mathematical output of the GPUs to accelerate the processing of large problems. Although similar in concept to CFD, these applications are much simpler to parallelize as they require limited message passing between computations. The first application relevant to CFD was in 2003 with the use of the Lattice Boltzmann particle method in OpenGL [23]. The LBM was perfectly suited to the massively parallel environment of GPUs due to the highly independent nature of the particle calculations. It was concluded that it was possible to accelerate the method relative to a CPU code by up to 50 times. However, OpenGL had its limitations and as graphics card were not specifically dedicated to calculations, the debugging was really difficult.

As it proved much easier and cost effective for industry to continue programming on traditional CPUs, which were continuing to get more and more powerful, NVidia realized its potential usefulness would be much greater if programmers could think of the GPU like a processor. NVidia selected a programming approach in which programmers would explicitly declare the data-parallel aspects of their workload. This didn’t come until late 2006 when NVidia released the first CUDA enabled graphics chipset -the G80. With the release of the SDK a few months later in early 2007, GPGPU became more user friendly, and it began expanding even further.

After the successful implementation of the Lattice Boltzmann code, developers from University of Cambridge began work on a 2D Euler Solver that would work on a GPU. In 2008 a traditional 2D/3D FORTRAN Euler Solver code was ported to an NVidia 8800GTX, using both BrookGPU (a high-level GPU programming language) and CUDA [7]. Huge speedups in both versions were reported, with 29 times for the 2D and 16 times for the 3D. However, it
was noted that the low memory of the graphics card (only 768MB) hindered the size of grid that could be kept on the GPU. This meant keeping the grid on the main RAM, and utilizing slow PCI-bus to transfer what was needed to the GPU, which had an important impact on performance. Regardless of this, however, the speedup compared to the serial CPU code is still impressive.

Considering its huge potential, a lot of studies have then been made on GPPU computing. We can for example mention Tolke and Krafczyk [30] who describe in 2008 a 3D Lattice Boltzmann model in detail with a CUDA implementation. Their single GPU implementation on an NVidia 8800 Ultra achieves a speedup of 100x over an Intel Xeon (noting that the CPU calculation was done in double precision and with an implementation of a more detailed model, making the speedup value not directly comparable). Simek et al. [27] details performance gains on a variety of single GPU platforms for atmospheric dispersion simulations, achieving speedups as high as 77x compared to a CPU implementation. Cohen and Molemaker [9] describe in 2009 the implementation and validation of an incompressible Navier-Stokes solver with Boussinesq approximation that supports double precision.

Since the memory on a graphics card is integrated and cannot be expanded, unlike conventional RAM, the simplest way to overcome the memory limitation was to use multiple cards in parallel. This line of thought lead to the development of a 3D Navier-Stokes solver integrating Message Passing Interface (MPI) and CUDA [16] (2010). This allowed the code to use a cluster of 64 CPU nodes with 2 Tesla GPUs each (128 GPUs in total), overcoming the memory issue by spreading the computation over several graphics cards. The code was a success, reporting accelerations of 130x over the CPU-only clusters on an 8 billion cell grid.

A number of issues related to floating point accuracy contribute to make the GPUs not very popular at first. Since they are designed primarily for graphical operations, where high precision is not of primary concern, GPU cores were originally only capable of single precision. This was elaborated on in 2011 [32], where it was found that older model GPU cores do not even abide by mathematical computation standards. Due to this and especially in CFD due to the many iterations performed, GPU codes can produce a final result that is not identical to that calculated by a CPU using an identical process. This issue was corrected with the modern hardware. Previously mentioned, the Boussinesq solver of Cohen et al. was implemented using the CUDA platform and is designed to run on the NVidia GT200 architecture for GPU computing. The NVidia Quadro FX5800 card which was used consists of a single GT200 GPU with 240 cores and 4GB of memory. GT200 supports IEEE-compliant double
precision math.

Despite the high cost, these new cards are capable of addressing both the precision issue and to some degree the memory problem. In 2011, Howe [15] proposed a 2D CFD code running on a new NVidia GTX480 via CUDA. This modern card supports twice the on-board memory of the old 8800GTX, with 1536MB, although it should be noted that this is still far less than the average RAM available on modern PCs. They found that although the precision issue had been corrected, with native double precision, there were still some odd quirks in the mathematical operations such as division taking several times longer to perform than multiplication. CPUs have this problem too, but most compilers will correct for this to reduce its impact, so it seems either CUDA has not evolved this ability yet or there is a more fundamental hardware issue.

Scientific computing with graphics processing units (GPU) has become a new paradigm in supercomputing. Some commercial software already allows the use of the power of GPU: NVidia and MathWorks have collaborated to GPU computing for MATLAB users. GPU acceleration enables faster results for users of the Parallel Computing Toolbox and MATLAB Distributed Computing Server.
Part III

Theory and numerical methods

1 Compressible Navier-Stoke equations

A flow obey to the general instantaneous Navier-Stokes equations. Those equations contains all the physics necessary to describe the motion of a fluid, including phenomena like turbulence or shocks. Those equations are generally given with comportment’s laws, used to close the system of equations, and state’s law, which describe the comportment of those variables.

1.1 Conservation equations

Generally speaking a fluid is characterized by its density $\rho = \rho(x, t)$ and its velocity $u(x, t) = (u_1, u_2, u_3)^T$, which are functions of the spatial variables $x = (x_1, x_2, x_3)^T$ and $t$. Using Einstein notation in cartesian coordinates, those density and velocity are linked by the mass conservation equation, or continuity equation:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_i}{\partial x_i} = 0$$ (3)

External volumic forces being neglected, the momentum conservation equation is given by:

$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} = \frac{\partial \sigma_{ij}}{\partial x_j} \quad i = 1, 2, 3$$ (4)

where $\sigma$ is the stress tensor. Finally the energy conservation equation is:

$$\frac{\partial \rho E}{\partial t} + \frac{\partial E \rho u_i}{\partial x_i} = \frac{\partial u_i \sigma_{ij}}{\partial x_j} - \frac{\partial \gamma_i}{\partial x_i}$$ (5)

with $E = e + \frac{1}{2} u^2$, sum of the $e$ internal energy and cinetic energy. Finally, $\gamma = (\gamma_1, \gamma_2, \gamma_3)^T$ represents the heat flux. This describes the macroscopic motion of all the fluids. However, for 5 equations we have 14 unknown : density, the 3 velocity components, the 6 symmetric stress tensor components, internal energy and the 6 heat flux components. Hence the need to introduce comportement’s law.
1.2 Comportements law

A fluid is called Newtonian if it can be described by linear laws giving:

- the stress tensor \( \sigma_{ij} \) as a function of the strain tensor \( s_{ij} \) (Newtons law):
  \[
  \sigma_{ij} = \left( -p + \lambda \frac{\partial u_l}{\partial x_l} \right) \delta_{ij} + 2\mu s_{ij}
  \] (6)

  where \( s_{ij} = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \) and \( \delta_{ij} \) is the identity tensor, also called Kroenecker symbol.

- and the heat flux \( \gamma_i \) as a function of the temperature gradient (Fourier's law):
  \[
  \gamma_i = -\kappa \frac{\partial T}{\partial x_i}
  \] (7)

The thermal conductivity can be expressed in function of the dynamic viscosity with Prandtl number as following:

\[
\text{Pr} = \frac{\mu}{\kappa} = \frac{\gamma \mu}{c_v}
\]

(8)

where \( c_p \) and \( c_v \) are the specific heat at constant pressure and volume, and \( \gamma = \frac{c_p}{c_v} \). Finally \( \mu \) and \( \lambda \) are linked by Stokes hypothesis:

\[
2\mu + 3\lambda = 0
\]

(9)

This introduces the 9 equations that were missing. It also introduces the properties of the fluid (dynamic viscosity \( \mu \) and volume viscosity \( \lambda \), thermal conductivity \( \kappa \)) that are supposed to be known, and two states variables \( p \) and \( T \). It is thus necessary to introduce other relations called states equations, that allow to have the evolution of those state variables.

1.3 States law

As we consider a calorically perfect gas, the equation of state provide the relation:

\[
p = \rho r T \quad \text{et} \quad e = c_v T
\]

(10)

where Mayer’s relation provides:

\[
r = c_p - c_v \quad \text{et} \quad \gamma = \frac{c_p}{c_v}
\]

(11)
As \( e = c_v T \), we can rewrite the heat flux with the following form:

\[
\gamma_i = -\kappa \frac{\mu c_v}{P_r} \frac{\partial T}{\partial x_i} = -\kappa \frac{\mu}{P_r} \frac{\partial T}{\partial x_i}
\]  

(12)

This achieve to close the system of the Navier-Stokes equations. However, aeronautical aerodynamics is characterized by compressible flows at high Reynolds numbers, so that the turbulence modeling is essential for predicting the flow field in agreement with experiments. A standard approach for this purpose is based on the so-called Reynolds-averaged Navier-Stokes (RANS) equations.

2 Averaging operators

2.1 Reynolds averaging

In a turbulent flow, velocity, pressure and temperature fluctuates in such a way that it becomes necessary to use a statistical approach. The flow is then decomposed in a mean part and in a fluctuating part \( f = \overline{f} + f' \), with \( \overline{f} \) the Reynolds average defined by:

\[
f(x, t) = \lim_{N \to +\infty} \left( \frac{1}{N} \sum_{n=1}^{N} f_n(x, t) \right)
\]

(13)

which signifies that \( N \) independent experiments are made on the same flow, and the variable \( f \) is stored at each position \( x \) and each time \( t \). Generally, this formulation is simplified and can be summed up by a time averaging:

\[
f(x) = \lim_{T \to +\infty} \frac{1}{T} \int_{0}^{T} f(x, t) \, dt
\]

(14)

2.2 Favre’s Averaging

Practically, the Reynolds decomposition is injected in the Navier-Stokes equations and then an equation for the mean flow is obtained. For example the decomposition of the density and the velocity is given by:

\[
\rho = \overline{\rho} + \rho' \quad \text{with} \quad \overline{\rho'} \neq 0
\]

\[
u = \overline{u} + u' \quad \text{with} \quad \overline{u'} \neq 0
\]
and for terms like $\rho u$, one would obtain:

$$\rho u = \rho u + \rho u' + \rho' u + \rho' u'$$

and then

$$\overline{\rho u} = \rho u + \rho' u'$$

This average is really difficult to use in compressible flows. Indeed, we want the averaged equations to keep the same form than in the incompressible case. A. Favre defined a new averaging operator (called weighted-average) given by:

$$\tilde{f} = \frac{\rho f}{\rho}$$

The fluctuating part is then given by

$$f'' = f - \tilde{f}$$

As the Reynolds average Favre’s operator is linear and idempotent:

$$\tilde{f}g = \tilde{f}\tilde{g} \quad \overline{fg} = \tilde{f}\tilde{g}$$

However it don’t commute woth spatial and temporal derivation operators. It is finally important to be noticed that:

$$\tilde{\tilde{u}} = \frac{1}{\rho} \left( \overline{\rho u} \right) = \tilde{u} \tilde{u}'' = 0 \quad (15)$$

and

$$\overline{\rho u} = \tilde{\rho} \tilde{u} \quad \overline{\rho u''} = 0 \quad (16)$$

### 3 Averaged Navier-Stokes Equations

This paragraph briefly explain how Averaged Navier-Stokes Equations. As seen before simple averages $\tilde{\phi}$ and mass weighted averages $\tilde{\phi}$ will be distinguished to simplify the equations. This decomposition is sometimes called Favre-Averaged Navier-Stokes equations. The reader should refer to for further information.
### 3.1 Continuity equation

Injecting the Reynolds decomposition in the mass conservation equation and taking part of its linearity properties gives:

\[
\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_j}{\partial x_j} = 0 \tag{17}
\]

Favre’s averaging then allow to write:

\[
\bar{\rho} u_j = \frac{\rho u_j}{\rho} = \rho \bar{u}_j \tag{18}
\]

which finally leads to:

\[
\frac{\partial \bar{\rho}}{\partial t} + \frac{\partial \bar{\rho} \bar{u}_j}{\partial x_j} = 0 \tag{19}
\]

Equation 19 clearly shows that Favre’s averaging allows the equation to keep the same form than the instantaneous one.

### 3.2 Momentum conservation equation

For this equation the same process than the continuity equation is used. First, the tensoriel product is decomposed in the following way:

\[
\bar{\rho} u_i u_j = \rho \left( \bar{u}_i + u''_i \right) \left( \bar{u}_j + u''_j \right)
= \bar{\rho} \bar{u}_i \bar{u}_j + \rho u''_i u''_j \tag{20}
\]

using the properties of Favre’s decomposition showed in equation 16. A supplementary term then appeared in the momentum equation, it will be modelled afterwards:

\[
\frac{\partial \bar{\rho} \bar{u}_i}{\partial t} + \frac{\partial \bar{\rho} \bar{u}_j \bar{u}_i}{\partial x_j} = \frac{\partial \sigma_{ij}}{\partial x_j} + \frac{\partial R_{ij}}{\partial x_j} \tag{21}
\]

where \( R_{ij} = -\bar{\rho} u_i u_j \) represents the Reynolds tensor. Besides, the Reynolds average stress tensor is given by:

\[
\sigma_{ij} = \left( -\bar{p} - \frac{2}{3} \mu \frac{\partial \bar{u}_l}{\partial x_l} \right) \delta_{ij} + 2 \mu \bar{s}_{ij} \tag{22}
\]
3.3 Energy conservation equation

As done before, introducing Reynolds and Favre decomposition leads to the following averaged energy conservation equation:

\[
\begin{align*}
\frac{\partial \rho \tilde{E}}{\partial t} + \frac{\partial \rho \tilde{E} \tilde{u}_j}{\partial x_j} = & -\frac{\partial \rho u''_j E''}{\partial x_j} + \frac{\partial \sigma_{ij} u_i}{\partial x_j} - \frac{\partial \gamma_j}{\partial x_j} \\
& (23)
\end{align*}
\]

According to the state law (10), the total energy can be written as:

\[
E = \frac{u^2}{2} + c_v T
\]

Thus, Favre’s average take the following form:

\[
\tilde{E} = \frac{u_i u_i}{2} + c_v \tilde{T} + \tilde{u}_i u''_i + \frac{1}{2} \tilde{u}'_i \tilde{u}''_i
\]

\[
= \frac{u_i u_i}{2} + c_v \tilde{T} + \tilde{u}_i u''_i
\]

\[
(25)
\]

Where \( k \) is known as the turbulent kinetic energy and is defined by:

\[
k = \frac{1}{2} \tilde{u}'_i \tilde{u}''_i
\]

\[
(26)
\]

For the state equation we will have:

\[
p = \rho r \tilde{T}
\]

\[
(27)
\]

Noticing that

\[
E'' = \frac{1}{2} (\tilde{u}_i + u''_i)(\tilde{u}_i + u''_i) - \tilde{u}_i \tilde{u}_i + c_v T''
\]

\[
= \frac{1}{2} u''_i u''_i
\]

\[
= \frac{1}{2} u''_i u''_i + \tilde{u}_i u''_i + c_v T''
\]

\[
(28)
\]

and neglecting third order terms finally leads for the term \( \rho u''_j E'' \) to:

\[
\rho u''_j E'' = \rho u''_j c_v T'' + \rho u''_j u''_j \tilde{u}_i
\]

\[
= \rho u''_j c_v T'' - R_{ij} \tilde{u}_i
\]

\[
(29)
\]

Where \( R_{ij} \) is the Reynolds stress tensor defined in equation (21).
Besides, according to equations (27) and (6) and Favre’s operator properties (16), taking into account Stokes hypothesis (9) and knowing that the tensorial product $u_i \delta_{ij} = u_j$, the term $\sigma_{ij} u_i$ is estimating in the following way:

$$\sigma_{ij} u_i = \left[ (-p - \frac{2}{3} \mu \frac{\partial u_i}{\partial x_i}) \delta_{ij} + 2 \mu s_{ij} \right] u_i$$

$$= -\rho u_j + \left[ \mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) \frac{2}{3} \mu \frac{\partial u_i}{\partial x_i} \delta_{ij} \right] u_i$$

$$= -\rho T u_j + \left[ \mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \frac{\partial u_i}{\partial x_i} \delta_{ij} \right] u_i$$

$$= -\rho T \tilde{u}_j - r \rho u_j T' + \left[ \mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \frac{\partial u_i}{\partial x_i} \delta_{ij} \right] u_i \quad (30)$$

At this point is generally supposed that the viscosity is sufficiently small to replace the operator $\cdot$ by the operator $\tilde{\cdot}$ in the third term of equation (30), even in the products:

$$\left[ \mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \frac{\partial u_i}{\partial x_i} \delta_{ij} \right] u_i \approx 2 \mu \tilde{s}_{ij} \tilde{u}_i - \frac{2}{3} \mu \tilde{u}_j \frac{\partial \tilde{u}_i}{\partial x_i} \quad (31)$$

The heat flux is treated the same way, which leads to:

$$\gamma_j = -\kappa \frac{\partial T}{\partial x_j} \approx -\kappa \frac{\partial \tilde{T}}{\partial x_j} \quad (32)$$

Finally using the relation $r = c_v (\gamma - 1)$, equations (29) and (30) allow to write the following equivalence:

$$- \rho u_j E' + \sigma_{ij} u_i = -\rho u_j c_v T' + R_{ij} \tilde{u}_i - r \rho T \tilde{u}_j - r \rho u_j T' + 2 \mu \tilde{s}_{ij} \tilde{u}_i - \frac{2}{3} \mu \tilde{u}_j \frac{\partial \tilde{u}_i}{\partial x_i}$$

$$= R_{ij} \tilde{u}_i - \gamma \rho c_v u_j T' + \tilde{u}_i \left[ (-p - \frac{2}{3} \mu \frac{\partial \tilde{u}_i}{\partial x_i}) \delta_{ij} + 2 \mu \tilde{s}_{ij} \right]$$

$$= R_{ij} \tilde{u}_i - \gamma \rho c_v u_j T' + \tilde{\sigma}_{ij} \tilde{u}_i \quad (33)$$

with the Favre average stress tensor defined as:

$$\tilde{\sigma}_{ij} = \left( -p - \frac{2}{3} \mu \frac{\partial \tilde{u}_i}{\partial x_i} \right) \delta_{ij} + 2 \mu \tilde{s}_{ij} \quad (34)$$
Using equations (33) and (32) leads to the Favre average energy equation:

\[
\frac{\partial \rho \bar{E}}{\partial t} + \frac{\partial \rho \bar{E} u_j}{\partial x_j} + \gamma \frac{\partial \rho c_v u''_j T''}{\partial x_j} = \frac{\partial \bar{\sigma}_{ij} u_i}{\partial x_j} + \frac{\partial R_{ij}}{\partial x_j} + \frac{\partial}{\partial x_j} \left( \kappa \frac{\partial \bar{T}}{\partial x_j} \right)
\]  
(35)

4 Boussinesq Hypothesis

The Reynolds stress tensor \( \bar{R} = -\rho\bar{u}'\otimes\bar{u}' \) correspond to a diffusion term in most of the fluid motions. In the RANS statistical approach, one have no access to the fluctuating variables such as \( u'' \) but the system need to be closed, though. In first order turbulence models, the Boussinesq hypothesis is generally used. This states that, by analogy with viscous stresses, the deviatoric part of the turbulent stresses are proportional to the strain of the mean flow through a dynamical eddy viscosity factor \( \mu_T \). This hypothesis can be written as followed:

\[
-\rho u''_i u''_j = \mu_T \left[ \frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i} - \frac{2}{3} \delta_{ij} \frac{\partial \bar{u}_l}{\partial u_l} \right] - \frac{2}{3} \rho k \delta_{ij}
\]  
(36)

where \( k = \frac{1}{2} \bar{u}_i'' \bar{u}_i'' \) is the turbulent kinetic energy defined in equation (26). This assumption makes sense in the way that turbulence generally bore in high gradients region (boundary layer, mixing layer...). However, there are some well-known situation in which this assumption is caught out: the border of attack of an aerofoil is laminar but gradients are though high. This leads to an over estimation of turbulence in this region.

The term \( \gamma \rho c_v u''_j T'' \) that appeared in the energy equation is also modeled in a close way to Boussinesq hypothesis. We will simply assume that:

\[
\gamma \rho c_v u''_j T'' = \kappa_T \frac{\partial \bar{T}}{\partial x_j} \quad \text{with} \quad \kappa_T = \frac{c_v \gamma}{P_T \mu_T}
\]  
(37)

where \( P_T \) is the turbulent Prandtl number. It will be fixed in all the study at 0.9.

Finally, assuming that the turbulent viscosity is sufficiently small to replace the operator \( \cdot \) by the operator \( \tilde{\cdot} \) in the Boussinesq hypothesis and using equations (19), (21), (35), (36) and (37), the Averaged Navier-Stokes complete system of equations can be summarized by:
\begin{align}
\frac{\partial \tilde{p}}{\partial t} + \frac{\partial \tilde{p} \tilde{u}_i}{\partial x_j} &= 0 \quad (38) \\
\frac{\partial \tilde{p} \tilde{u}_i}{\partial t} + \frac{\partial \tilde{p} \tilde{u}_j \tilde{u}_i}{\partial x_j} &= \frac{\partial \tilde{\sigma}^*_{ij}}{\partial x_j} \quad (39) \\
\frac{\partial \tilde{p} \tilde{E}}{\partial t} + \frac{\partial \tilde{p} \tilde{E} \tilde{u}_j}{\partial x_j} &= \frac{\partial \tilde{\sigma}^*_{ij} \tilde{u}_i}{\partial x_j} - \frac{\partial \tilde{\gamma}^*_j}{\partial x_j} \quad (40)
\end{align}

where the tensor $\tilde{\sigma}^*_{ij}$ and the vector $\tilde{\gamma}^*_j$ are written:

\begin{align}
\tilde{\sigma}^*_{ij} &= -\tilde{p}^* \delta_{ij} + 2(\mu + \mu_T) \left[ \tilde{s}_{ij} - \frac{1}{3} \frac{\partial \tilde{u}_l}{\partial x_l} \delta_{ij} \right] \quad (41) \\
\tilde{\gamma}^*_j &= -\gamma_c \left( \frac{\mu}{\Pr} + \frac{\mu_T}{\Pr_T} \right) \frac{\partial \tilde{T}}{\partial x_j} \quad (42) \\
\tilde{p}^* &= \bar{p} + \frac{2}{3} \bar{p}^* \quad (43)
\end{align}

In most of the case the contribution of turbulent kinetic energy $k$ to the pressure term can be neglected. As a consequence, a complete model can be obtained by writing a transport equation for $\mu_T$. A lot of models try to propose such an equation. Among those, the model proposed by P.R. Spalart and S.R. Allmaras, in 1992, [28] is presented in the following section.

## 5 Spalart-Allmaras model

### 5.1 Motivations

The simplest models for turbulence are \textit{algebraic models} or \textit{zero equation models}. Those models don’t imply the resolution of a differential equation, and thus are relatively easy to implement. However, they offer a very poor precision and are not well-suited for aeronautical computations. Moreover those models are not complete, is the way they required an \textit{a priori} knowledge of the flow.

Two equations models such as the well-known $k - \varepsilon$ model or $k - \omega$ model present a most sophisticated way of closing the system of equations. In the case of $k - \varepsilon$ an equation is generally written for the turbulent kinetic energy $k$ and for the dissipation rate $\varepsilon$. But the computational cost is far higher than algebraic models, and sometimes needs a specific treatment of boundary.
Spalart-Allmaras model - a *one equation model* - proposes a transport equation for the turbulent viscosity. It is the lowest level at which a model can be complete. It was originally developed for aeronautical applications to remove incompleteness from algebraic and yet have a simpler model than two equations described before. It is even today often used in Aeronautic industry. Indeed, it is easy to implement on unstructured grid and quite robust, as it don’t require a particular fine grid near the walls. This allows a faster convergence to the solution and a low memory treatment of turbulence. One must not forget that it has to be implemented on GPU, where memory requirement is much more important than on CPU. SA model is finally the best compromise between completeness of the model, accuracy of the results and time/memory cost of the solution.

### 5.2 General incompressible formulation

The evolution of turbulent viscosity is modeled by a partial differential equation, containing a diffusion, production and destruction term. Generally speaking when a scalar variable $F$ obey to a conservation law it can be put in the following form:

$$\frac{\partial F}{\partial t} + \mathbf{u} \cdot \nabla F = \text{Diffusion} + \text{Production} + \text{Destruction} \quad (44)$$

where $\mathbf{u}$ is the convective velocity. To create a model it is thus necessary to define the expression of those diffusion, production and destruction terms. This will lead to the introduction of some constant. To define them P.R. Spalart and S.R. Allmaras imposed to the model to be coherent with some experimental and numerical results. This procedure is known as calibration.

#### 5.2.1 Diffusion term

Classical diffusion operator are of the form:

$$\nabla \cdot \left( \frac{\nu_t}{\sigma} \nabla \nu_t \right)$$

where $\nu_t = \mu T/\rho$ represents the kinematic turbulent viscosity and $\sigma$ is a constant. The molecular viscosity is introduced and a non-conservative part is then added which finally lead to:

$$\text{Diffusion} = \frac{1}{\sigma} \left[ \nabla \cdot ((\nu + \nu_t)\nabla \nu_t) + c_b (\nabla \nu_t)^2 \right] \quad (45)$$

35
where again $c_{b_2}$ is a constant which will be discussed later.

### 5.2.2 Production term

Spalart and Allmaras based the construction of the production term on the vorticity $S = \sqrt{2\Omega_{ij}\Omega_{ij}}$ where $\Omega_{ij} = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} - \frac{\partial u_j}{\partial x_i} \right)$ is the rotation tensor. Indeed, in an aeronautical context turbulence is found where vorticity is, both emanating from the solid boundaries. The production term is thus written:

$$\text{Production} = c_{b_1} S \nu_t$$  \hspace{1cm} (46)

where $c_{b_1}$ is a constant.

### 5.2.3 Destruction term

In the near wall region some particular physical phenomena take place. For example the blocking effect is a consequence of the incompressibility of the fluctuating velocities. Any fluctuation in the direction of the wall generates a zone of high pressure that leads to a selective damping of the wall-normal fluctuation. This effect acts as the main destruction term for Reynolds shear stress and is non local: it is felt at distance from the wall through the pressure term.

This suggests to Spalart and Allmaras a destruction term in the transport equation for eddy viscosity. Dimensionnal analysis leads to a combination $-c_{w_1}(\nu_t/d)^2$, with $d$ the distance to the wall. This term will be passive in free shear flow (far from the wall, $d \to \infty$). Experiments on flate pates however shows that this destruction term decays too slowly in the outer region of the boundary layer. Thus, they multiply the term by a non-dimensional function $f_{w_1}$ that will be discussed later. Introducing the constant $c_{w_1}$ leads to:

$$\text{Destruction} = -f_{w_1} c_{w_1} \left( \frac{\nu_t}{d} \right)^2$$ \hspace{1cm} (47)

### 5.2.4 Calibration of the model

#### Free shear flow

The constants $\sigma$, $c_{b_1}$ and $c_{b_2}$ are calibrated by requiring correct level of shear stress in two dimensional mixing layer and wakes. Peak shear stress values can be considered respectively as $0.01(\Delta U)^2$ and $0.06(\Delta U)^2$, where $\Delta U$ is the peak velocity difference. This gives two conditions
on the three constants. Based on experimental results Spalart and Allamaras then proposed:

$$\sigma = \frac{2}{3} \quad c_{b_1} = 0.1355 \quad c_{b_2} = 0.6220$$

(48)

**Near wall region**

First, we present a brief recall of the structure of a classical turbulent boundary layer:

![Velocity profile](image)

Figure 7: Velocity profile of a turbulent boundary layer. Theoretical linear and logarithmic profiles and DNS data.

$$u^+ = \frac{u}{u_\tau} \quad y^+ = \frac{y u_\tau}{\nu} \quad \rho u_\tau^2 = \tau_w$$

where $\tau_w = \mu \frac{du}{dy}|_0$ is the friction at the wall. The boundary layer can thus be divided into three main parts. The first one is the viscous sublayer ($y^+ < 5$) in which:

$$u^+ = y^+$$

For about $5 < y^+ < 30$ the buffer layer allows a transition to the logarithmic layer going from $y^+ \approx 30$ to $y^+ \approx 1000$ where:

$$u^+ = \frac{1}{\kappa} \ln(y^+) + C$$

where $\kappa$ and $C$ are constants.
In the logarithmic zone, vorticity and turbulent viscosity are given respectively by
\[ S = \frac{u_t}{(\kappa d)} \text{ and } \nu_t = u_t \kappa d \] where \( d \) is the distance to the wall. Writing the equilibrium between the production, diffusion and destruction term leads to the following relation:

\[ c_{w_1} = \frac{c_{b_1}}{\kappa^2} + \frac{1 - c_{b_2}}{\sigma} \] (49)

Moreover, the choice of the function \( f_{w_1} \) is inspired by the algebraic models, in which the mixing length play an important role near the wall. This length is defined as \( l = \sqrt{\nu_t/S} \) and \( l/\kappa d \) is used as a non dimensional parameter. This defines :

\[ r = \frac{\nu_t}{S \kappa^2 d^2} \] (50)

A satisfactory function had then been proposed by Spalart and Allmaras :

\[
\begin{align*}
 f_{w_1} &= g \left( \frac{1 + c_{w_3}^6}{g^6 + c_{w_3}^6} \right)^{1/6}, \quad g = r + c_{w_2} (r^6 - r) \quad c_{w_2} = 2, \quad c_{w_3} = 0.3
\end{align*}
\] (51)

Due to the buffer layer and viscous sublayer additional notation have been introduced: \( \hat{\nu} \) which is equal to \( \nu_t \) except in the viscous region and \( \chi = \hat{\nu}/\nu_t \). \( \hat{\nu} \) is then defined by :

\[
\begin{align*}
 \nu_t &= f_{v_1} \hat{\nu}, \quad f_{v_1} = \frac{\chi^3}{\chi^3 + c_{v_1}^3}, \quad c_{v_1} = 7.1
\end{align*}
\] (52)

The introduction of this new variable leads to a modification of the production term. The vorticity os the flow is now written:

\[
\begin{align*}
 \hat{S} = S + \frac{\hat{\nu}}{\kappa^2 d^2} f_{v_2} \quad f_{v_2} = 1 - \frac{\chi}{1 + \chi f_{v_1}}
\end{align*}
\] (53)

All the terms will finally be written with \( \hat{\nu} \) instead of \( \nu_t \), like for example \( r = \frac{\nu_t}{S \kappa^2 d^2} \).

### 5.2.5 Final formulation

The incompressible Spalart-Allmaras model can finally be written:

\[
\begin{align*}
 \frac{\partial \hat{\nu}}{\partial t} + \overline{u_i} \frac{\partial \hat{\nu}}{\partial x_i} = P(\hat{\nu}) + D(\hat{\nu}) + M(\hat{\nu})
\end{align*}
\] (54)
where $P(\hat{\nu})$ is the production term:

$$P(\hat{\nu}) = c_b \hat{S} \hat{\nu} \quad (55)$$

$D(\hat{\nu})$ is the destruction term:

$$D(\hat{\nu}) = -c_{w_1} f_{w_1} \left( \frac{\hat{\nu}}{d} \right)^2 \quad (56)$$

$M(\hat{\nu})$ is the diffusion term:

$$M(\hat{\nu}) = \frac{1}{\sigma} \left[ \frac{\partial}{\partial x_j} \left( (\nu + \hat{\nu}) \frac{\partial \hat{\nu}}{\partial x_j} \right) + c_b \frac{\partial \hat{\nu}}{\partial x_i} \frac{\partial \hat{\nu}}{\partial x_i} \right] \quad (57)$$
5.3 Compressible Spalart-Allamaras model

A lot of works [5] extented to compressible flows the model described previously by tranporting the quantity $\rho \hat{\nu}$ instead of $\hat{\nu}$. Another solution has been proposed by Catris and Aupoix [8] by modifying the diffusion laws in the turbulence model. Catris suggests to advect $\rho \hat{\nu}$ and to diffuse $\rho \sqrt{\hat{\nu}}$. Nevertheless, this strategy complicates the numerical implementation. More recently Steven Allmaras [3] tried to clarify the confusion around compressible SA: “Confusion exists in the literature over the formulation of S-A for compressible flows. We reaffirm that the formulation presented above is applicable to both incompressible and compressible flows, and it should be considered the standard form for compressible.”. The final formulation adopted in this code will then follow the original paper [28] :

\[
\frac{\partial \hat{\nu}}{\partial t} + \mathbf{u}_i \frac{\partial \hat{\nu}}{\partial x_i} = c_{b1} (1 - f_{t2}) \hat{S} \hat{\nu} + \frac{1}{\sigma} \left[ \frac{\partial}{\partial x_j} \left( (\nu + \hat{\nu}) \frac{\partial \hat{\nu}}{\partial x_j} \right) + c_{b2} \frac{\partial \hat{\nu}}{\partial x_i} \frac{\partial \hat{\nu}}{\partial x_i} \right] - \left[ c_{w1} f_{w1} - \frac{c_{b1}}{\kappa^2} f_{t2} \right] \left( \frac{\hat{\nu}}{\hat{d}} \right)^2
\]

where

\[
\sigma = 2/3 \quad c_{b1} = 0.1355 \quad c_{b2} = 0.6220
\]

\[
c_{w1} = \frac{c_{b1}}{\kappa^2} + \frac{1 - c_{b2}}{\sigma} \quad r = \frac{\hat{\nu}}{\hat{S} \kappa^2 d^2}
\]

\[
f_{w1} = \frac{1 + c_{w2}^6}{g^6 + c_{w3}^6} \left( f_{w1} \hat{\nu} + \hat{S} \hat{S} \right) = r + c_{w2} (r^6 - r) \quad c_{w2} = 2 \quad c_{w3} = 0.3
\]

\[
\nu_t = f_{v1} \hat{\nu} \quad \hat{S} = \frac{\hat{\nu}}{\kappa^2 d^2} \quad \chi = \frac{\hat{\nu}}{\nu} \quad f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3} \quad f_{v2} = 1 - \frac{\chi}{1 + \chi f_{v1}} \quad c_{v1} = 7.1
\]

\[
f_{t2} = c_{t3} \exp (-c_{t4} \chi^2) \quad c_{t3} = 1.2 \quad c_{t4} = 0.5
\]

The introduction of the function $f_{t2}$ is to be noted. The function was introduced by Spalart and Allmaras to provide control over the laminar regions of the shear layers. It has two main
aspects: keeping the flow laminar where desired and obtaining transition when desired. For a complete description of the function see [28].

6 Numerical methods

In this part, the notation $\tilde{\cdot}$ and $\bar{\cdot}$ of the Favre and Reynolds average will be intentionally deleted to simplify the comprehension. Besides, we will consider a purely two dimensional flow.

6.1 Problem description

To simplify the problem, the complete and close system of equation (Navier-Stokes equations (38)-(40) and Spalart-Allamaras turbulence model (58)) can be written in a vectorial form:

$$\frac{\partial W}{\partial t} + \frac{\partial F_i^c}{\partial x_i} - \frac{\partial F_i^v}{\partial x_i} = S$$

(59)

where $\mathbf{U}$ is the vector of the conservative variables and $\mathbf{F}_i^c$, $\mathbf{F}_i^v$ and $S$ are respectively the convective, viscous ans source terms of the equation. They are defined as:

$$\mathbf{W} = \begin{pmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho E \\ \rho \hat{\nu} \end{pmatrix} \quad \mathbf{F}_i^c = \begin{pmatrix} \rho u_i \\ \rho u_i u_1 + \delta_{i1} p \\ \rho u_i u_2 + \delta_{i2} p \\ u_i (\rho E + p) \\ \rho u_i \hat{\nu} \end{pmatrix} \quad \mathbf{F}_i^v = \begin{pmatrix} 0 \\ \tau_{i1} \\ \tau_{i2} \\ \gamma_i + \tau_{ik} u_k \\ V_i(\hat{\nu}) \end{pmatrix} \quad \mathbf{S} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

where $\rho$ is the density, $u_i$ the velocity components, $E$ total energy, $p$ the pressure and $T$ the temperature. The viscous tensor $\tau_{ij}$ and the heat flux $\gamma_i$ are defined, according to relations (41) and (42), by:

$$\tau_{ij} = 2(\mu + \mu_T) \left[ s_{ij} - \frac{1}{3} \frac{\partial u_i}{\partial x_i} \delta_{ij} \right]$$

(60)

$$\gamma_i = -\gamma c_v \left( \frac{\mu}{Pr} + \frac{\mu_T}{Pr_T} \right) \frac{T}{\partial x_i}$$

(61)

and the terms $V_i(\hat{\nu})$ and $S_i(\hat{\nu})$ are defined, according to equation (58):
\[ V_i(\hat{\nu}) = -\frac{1}{\sigma} \left( \rho(\nu + \hat{\nu}) \frac{\partial \hat{\nu}}{\partial x_i} \right) \]  

\[ S(\hat{\nu}) = c_b(1 - f_t)\rho \hat{\nu} + \frac{1}{\sigma} \left[ c_b \rho \frac{\partial \hat{\nu}}{\partial x_i} \frac{\partial \hat{\nu}}{\partial x_i} - \left[ c_w f_{w_1} - \frac{c_b}{\kappa^2 f_t} \right] \rho \left( \frac{\hat{\nu}}{d} \right)^2 \right] \]  

For the dependence of the laminar viscosity on temperature, Sutherland's law is used:

\[ \mu(T) = \mu_0 \left( \frac{T}{T_0} \right)^{3/2} \cdot \frac{T_0 + 110.4}{T + 110.4} \]  

with \( T_0 = 273.16 \, \text{K} \) and \( \mu_0 = 1.711 \cdot 10^{-5} \, \text{kg m}^{-1} \text{s}^{-1} \).

### 6.2 Finite volume formulation

Let's now integrate equation (59) over a control volume \( \Omega \), as illustrated on figure 8:

\[ \int_{\Omega} \frac{\partial \mathbf{W}}{\partial t} d\Omega + \int_{\Omega} \frac{\partial}{\partial x_i} \left( \bar{\mathbf{F}}^c - \bar{\mathbf{F}}^v \right) d\Omega = \int_{\Omega} S d\Omega \]  

(65)

Let's note, for better comprehension, \( \bar{\mathbf{F}}^c = T(\mathbf{F}^c_1, \mathbf{F}^c_2) \) and \( \bar{\mathbf{F}}^v = T(\mathbf{F}^v_1, \mathbf{F}^v_2) \) so that equation (65) can be written:

\[ \int_{\Omega} \frac{\partial \mathbf{W}}{\partial t} d\Omega + \int_{\Omega} \nabla \cdot (\bar{\mathbf{F}}^c - \bar{\mathbf{F}}^v) d\Omega = \int_{\Omega} S d\Omega \]  

(66)

Green-Ostrogradsky theorem then allows to rewrite equation (83) in the following form, where \( \partial \Omega \) is the boundary surface of the control volume \( \Omega \) :

\[ \int_{\Omega} \frac{\partial \mathbf{W}}{\partial t} d\Omega + \int_{\partial \Omega} (\bar{\mathbf{F}}^c - \bar{\mathbf{F}}^v) \cdot \mathbf{dS} = \int_{\Omega} S d\Omega \]  

(67)

If applied to an unstructured grid, comporting control volumes delimited by \( N \) edges (figure 9), and assuming that all the conservative variables defined previously can be considered as constant over one control volume, equation (67) can be written:

\[ V_\Omega \frac{\partial \mathbf{W}}{\partial t} + \sum_{n=1}^{N} (\bar{\mathbf{F}}^c_{(n)} - \bar{\mathbf{F}}^v_{(n)}) \cdot \Delta \mathbf{S}_{(n)} = V_\Omega \mathbf{S} \]  

(68)
where $\bar{F}^c_{(n)}$ represents the value of vector $\bar{F}^c$ along edge $n$ and $\Delta S_{(n)}$ the normal to this edge. The full discretization of the second and third terms of the previous equation is usually denoted the residual $R$, so that equation (68) can be written:

$$V_\Omega \frac{\partial W}{\partial t} = -R_\Omega$$ (69)

### 6.2.1 Space discretization

The values of the flow variables are approximated as constants on the control-volumes, resulting in discontinuities at the cell interfaces. Hence $\bar{F}^c$ and $\bar{F}^v$ must be approximated by numerical fluxes $\hat{F}^c$ and $\hat{F}^c$ in the surface integral of equation (68). For convenience we will now denote by index $a$ the cell considered and by $b$ the neighbour cell. The following obvious second order central discretization scheme is then used:

$$\bar{F}^c_{(n)} = \bar{F}^c_{ab} = \frac{1}{2}(\bar{F}^c_a + \bar{F}^c_b)$$ (70)

This scheme shown himself unconditionally unstable if used in this state. However Jameson, Schmidt and Turkel had the idea to simply add an artificial dissipation term to this classical central scheme [17] to make it stable. This is the strategy used here and it will be discussed later.
**Evaluation of the gradients:**

Some of the numerical fluxes components, especially the viscous ones, involved first order derivative in respect to space. Those gradients have then to be evaluated on the cell faces. A general procedure can be derived by the use of Green-Ostrogradsky. Since for an arbitrary control volume $\Omega$

$$\int_{\Omega} \nabla U d\Omega = \oint_{\partial\Omega} U dS \quad (71)$$

the averaged gradients can be defined as:

$$\left( \frac{\partial U}{\partial x} \right)_\Omega = \frac{1}{V_\Omega} \int_{\Omega} \frac{\partial U}{\partial x} d\Omega = \oint_{\partial\Omega} U e_x \cdot dS \quad (72)$$

Considering now the situation of figure 10, we want to evaluate the gradient of $U$ at the cell face. We then consider the auxiliary control volume $\Omega_{aux}$:

**Figure 11:** Auxiliary control volume defined to compute gradients at cell faces

and equation (72) leads to

$$\left( \frac{\partial U}{\partial x} \right)_{ab} = \frac{1}{V_{\Omega_{aux}}} \oint_{\partial\Omega_{aux}} U dy = -\frac{1}{V_{\Omega_{aux}}} \oint_{\partial\Omega_{aux}} ydU$$

$$= \frac{1}{V_{\Omega_{aux}}} \sum_i \frac{1}{2} (U_I + U_{I+1})(y_{I+1} - y_I) \quad (73)$$

where index $I$ represents the node that constitute the auxiliary cell.

**6.2.2 Time integration**

Instead of simply advancing in time using a classical forward explicit method:

$$W^{n+1} = W^n - \frac{\Delta t}{V_\Omega} R_\Omega \quad (74)$$
an explicit scheme 4 step Runge-Kutta method is used to advance in time. The basic idea of Runge Kutta (RK) method is to evaluate the right hand side of the system (69) at several values of $W$ in the interval between $n\Delta t$ and $(n + 1)\Delta t$, and to combine them in order to obtain a high order approximation of $W^{(n+1)}$. The number of intermediate values is referred to as the Runge Kutta stages.

\[
W^{(1)} = W^n - \frac{\Delta t}{V_\Omega} \alpha_1 R_\Omega^n
\]
\[
W^{(2)} = W^n - \frac{\Delta t}{V_\Omega} \alpha_2 R_\Omega^{(1)}
\]
\[
W^{(3)} = W^n - \frac{\Delta t}{V_\Omega} \alpha_3 R_\Omega^{(2)}
\]
\[
W^{n+1} = W^n - \frac{\Delta t}{V_\Omega} \alpha_4 R_\Omega^{(3)}
\]

(75)

with

\[
\alpha_1 = \frac{1}{4}, \quad \alpha_2 = \frac{1}{3}, \quad \alpha_3 = \frac{1}{2}, \quad \alpha_4 = 1
\]

(76)

As we are not interested in the transient behaviour of the solution, a local time step can be choose, so that each cell progress at its maximum possible time step. This makes loose the time consistency of the solution but allow significant convergence acceleration. The local inviscid local time step is calculating as follows:

\[
\delta t = CFL \frac{V_\Omega}{\sum_{edges} |n \cdot u + c \sqrt{dx^2 + dy^2}|}
\]

(77)

The CFL number has to be chosen under the stability condition of the 4 stages RK time scheme. This scheme shown himself stable for $CFL < 2.8$. A viscous local time step is also calculated. For the complete Navier-Stokes equation the local time step to be applied is the minimum between the inviscid and the viscous time steps.

6.3 Artificial dissipation

Artificial dissipation is required to prevent oscillatory results in regions of high pressure gradient, such as near shockwaves. The scheme used in this code was described by Jameson, Schmidt and Turkel [17] and involves the use of Dissipation Constants that can be tailored to
the type of problem being run.

\[ D_i = \varepsilon_i^{(2)}(d_i) + \varepsilon_i^{(4)} \left( \sum_{\text{edges}} d_i - d_j \right) \]  

(78)

where

\[ d_i = \sum_{\text{edges}} W_i - W_j \]  

(79)

\[ d_i = \sum_{\text{edges}} W_j - W_i \]  

(80)

Where \( i \) is the current cell, \( j \) is the interfacing cell (illustrated as cell a and b in Figure 10), \( W \) contains the cell centre values defined at in equation 59, \( \varepsilon_i^{(2)} \) and \( \varepsilon_i^{(4)} \) are defined below:

\[ \varepsilon_i^{(2)} = \kappa^{(2)} \max(v_j, v_i) \]  

(81)

\[ \varepsilon_i^{(4)} = \max(0, (\kappa^{(2)} - \varepsilon_i^{(2)})) \]  

(82)

And \( v \) is defined as:

\[ v_i = \frac{|p_j - 2p_i + p_{i-1}|}{|p_j| + 2|p_i| + |p_{i-1}|} \]  

(83)

Here \( i - 1 \) and \( j + 1 \) are the cells before \( i \) and beyond \( j \) respectively, and \( \kappa^{(2)} \) and \( \kappa^{(4)} \) are the Dissipation Constants. For this code, the values will be set at \( \frac{1}{2} \) and \( \frac{2}{125} \) respectively. Within the code this will be done in a two step process the first loop will calculate the differences in the conserved variables required in equations 79 and 80, storing the data in an array labelled \( \text{dw2} \) for use in the second loop, which will complete the calculation in equation 78 and store the final dissipation values in the final array \( \text{dis} \).
Part IV

GPU implementation

1 Hardware

Originally a Graphics Processing Unit or GPU (also occasionally called Visual Processing Unit or VPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory in such a way so as to accelerate the building of images in a frame buffer intended for output to a display.

Figure 12 shows the architecture of a typical CUDA-capable GPU (here a G80). It is organized into an array of highly threaded Streaming Multiprocessors (SMs). Each SM is made of a group of Stream Processors (SPs), which are the GPU cores. Each GPU currently comes with up to 1 to 4 gigabytes of graphics double data rate (GDDR) DRAM, referred to as global memory or Video-RAM (VRAM) in Figure 12. Note how each Streaming Multiprocessor has its own shared memory and cache, but all have access to the VRAM. For graphics applications, it hold video images, and texture information for three-dimensional (3D) rendering, but for computing it functions as very-high-bandwidth, off-chip memory, though with more latency than typical system memory. For massively parallel applications, the higher bandwidth makes up for the longer latency.

Figure 12: CPUs and GPUs have fundamentally different design philosophies
Access to the shared and cache memory by cores in SMs is faster than accessing the VRAM directly, so data to be used in that SM should be loaded onto the shared memory for the more effective use. All this will be discussed later in an optimization section. The massively parallel G80 chip has 128 SPs (16 SMs, each with 8 SPs). But the most recent GPUs dedicated to general computing by Nvidia has more SPs. In table 17 are given the properties of the Tesla C2050 used in all this study. The number of threads and blocks will be discussed in the followings sections.

<table>
<thead>
<tr>
<th>NVidia Tesla C2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total amount of global memory</td>
</tr>
<tr>
<td>Number of Streaming Multiprocessors</td>
</tr>
<tr>
<td>Number of Stream Processors per SM</td>
</tr>
<tr>
<td>Total</td>
</tr>
<tr>
<td>GPU Clock Speed</td>
</tr>
<tr>
<td>Total amount of constant memory</td>
</tr>
<tr>
<td>Total amount of shared memory per block</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
</tr>
<tr>
<td>Maximum sizes of each dimension of a block</td>
</tr>
<tr>
<td>Maximum sizes of each dimension of a grid</td>
</tr>
</tbody>
</table>

Table 2: Summary of the Tesla C2050 properties

2 GPU program structure

Within the CUDA environment, there are two main entities: the Host and the Device. The Host is the name given to the CPU environment, which utilises the RAM as its memory allocations. The Device is the GPU, which accesses the VRAM for its memory allocations. It is very important to keep track of which entity data is on at any given point, as they are completely isolated from each other. The host cannot operate on data held on the device memory and vice versa. A CUDA code is a list of CPU instructions. Among them, there are several call to kernels which referred to pieces of code that are executed on the GPU to manipulate data contained on the VRAM. The following picture shows the basic structure of a CUDA program:
Figure 13: Basical execution of a CUDA program

More precisely all codes written with CUDA usually follow the same basic structure:

- Initiate all CPU arrays
- Initiate all GPU arrays
- Calculate size of GPU arrays
- Allocate GPU arrays with calculated size
- Fill all arrays with data
- Copy arrays required by the kernel(s) to the device memory
- Set up the parallel environment
- Execute the kernel(s)
- Copy arrays back to host memory
- Clear device memory

Once the arrays have been copied back to the host, they can be operated on by the CPU again in any required way.
3 Grids, Blocks and Threads

Before executing a kernel on a GPU, the parallel environment must be initialised. This environment is split into Grids, Blocks and Threads, as illustrated in Figure 14. Each grid contains a number of blocks, and each block contains a number of threads, which in turn execute the kernel. Each level in this hierarchy has its own fast-access memory: VRAM for the whole grid, shared memory for each block and registers for each thread. Basically a block is executed on a SM, and each threads of a block is in turn executed on a SP. As we’ll see the memory locality can be exploited to optimise the code, by preloading relevant data to the SM shared memory to allow quicker access than from the VRAM.

Most importantly though, contents of each grid (the number of blocks and threads within them) can be specified by the programmer on a per-kernel basis. This is done with the CUDA functions dimGrid and dimBlock, which allows the number of blocks to be specified in terms of a 2D array, and the threads as a 3D array. Each element in these arrays has its own unique ID value, which differentiates it from all the other threads and blocks. This is also demonstrated in Figure 14. These values can be accessed via the blockIdx and threadIdx functions, and thus exploited by the programmer to specify what data each kernel deals with (see following section for an example).

Figure 14: Layout of Grids, Blocks and Threads
4 Example on a Matrix/Vector multiplication

To illustrate the use of GPU in General Purpose Parallel Computing, we will here discuss a simple example of a Matrix Vector multiplication program. The full programme can be found in annex.

1 - Initiate CPU arrays
This is done through the classical malloc function. Here, the matrix P will be of size $N \times N$, vector Q and R of size $N$. All in double precision type.

\[
\begin{align*}
P &= (\text{double}*)\text{malloc}(N*\text{sizeof(double)}); \\
Q &= (\text{double}*)\text{malloc}(N*\text{sizeof(double)}); \\
R &= (\text{double}*)\text{malloc}(N*\text{sizeof(double)}); \\
\end{align*}
\]

2 - Initiate GPU arrays
This is done through the function cudamalloc, extension from CUDA to the previous function. The variables located on the device will generally be denoted with the prefix d_.

\[
\begin{align*}
cudamalloc((\text{void}**)&d_P, N*N*\text{sizeof(double)}); \\
cudamalloc((\text{void}**)&d_Q, N*\text{sizeof(double)}); \\
cudamalloc((\text{void}**)&d_R, N*\text{sizeof(double)}); \\
\end{align*}
\]

3 - Fill CPU arrays with data
This is usually an input of the user, only P and Q are filled, as R is kept to store the result.

4 - Transfert data to GPU
This is done through the function cudaMemcpy. The flag cudaMemcpyHostToDevice indicates that the data have to be transferred from CPU to GPU. First argument is the array in which data will be copied, and second argument is the array to copy. The third one is the size of the data. Only P and Q are transferred as d_R will store the result on the GPU.

\[
\begin{align*}
cudacopy(d_P, P, N*N*\text{sizeof(double)}, \text{cudaMemcpyHostToDevice}); \\
cudacopy(d_Q, Q, N*N*\text{sizeof(double)}, \text{cudaMemcpyHostToDevice}); \\
\end{align*}
\]

5 - Set up parallel environment
The vector is constituted of $N$ elements. The idea of CUDA environment is to divide those
$N$ elements in $M_b$ number of blocks, each one constituted of $M_t$ number of threads. When a kernel is then launched on the GPU, each SM will concurrently run each block, when at the same time each SP of a SM will concurrently run different threads. Figure ?? show a basic illustration of the principle on a $4 \times 4$ matrix.

$M_b$ and $M_t$ are declared as the CUDA type `dim3`. As indicated in table ?? the maximum number of threads per block is 1024. We will supposed for convenience that $N$ is a multiple of 1024 and assigned 1024 threads per block, which leads to $N/1024$ blocks.

$$\begin{align*}
\text{dim3 } M_b(N/1024,1); \\
\text{dim3 } M_t(1024,1,1);
\end{align*}$$

6 - Execution of the kernel

A kernel is a function called by the host and executed on the device. From the CPU point of view, we will have a simple call to a kernel called `matmul`

```c
matmul<<<M_b,M_t>>>(d_P,d_Q,d_R,N);
```

where between the brackets are specified the size of the grid and blocks defined previously. From the GPU point of view the kernel is written as follows:

```c
__global__ (double *d_P,double *d_Q,double *d_R,int N) {
```

the identifier `__global__` specified that this function is a kernel that can only be run on GPU and called from CPU (no recursive calls are allowed on GPU).

```c
int i = threadIdx.x + (blockIdx.x * blockDim.x);
```

As explained in section 3, this is the most important declaration of a kernel. It allows indeed to locate precisely the thread executing the kernel, and will allow each thread to treat different data while running the same kernel. `threadIdx.x` represent the index of a thread within a block, whereas `blockIdx.x` represents the index of a block on the GPU. `blockDim.x` simply represent the number of thread in a block (here 1024). In this context $i$ will represent the raw of the computed vector. The rest of the kernel only consists in the basic matrix/vector multiplication algorithm:
```c
int k;
double elem = 0.0;
for(k=0;k<N;k++) {
    elem += d_P[i*N+k]*d_Q[k];
}
d_R[i] = elem;
```

7 - *Copy arrays back and clear*

As is part 4, the results need then to be transferred back to the CPU, to print in a file or for any post treatment. It is done the same way than transferring to the GPU, but using the flag `cudaMemcpyDeviceToHost`. Then it is important to clear the memory of the GPU, by using the function `cudaFree`.

```c
    cudaMemcpy(R, d_R, N*sizeof(double), cudaMemcpyDeviceToHost);
cudaFree(d_R);
cudaFree(d_Q);
cudaFree(d_P);
```

The full code can be found in annex A. To give a first overview of the capability of the GPU figure 15 shows the time execution of both CPU and GPU in function of the size of the matrix P. For convenience, the time of 10 products has been measured:
NVidia Tesla C2050 is the GPU that will be used in our study, NVidia GeForce GT520 MX, is a bottom of the range GPU chip that can be found on most of the latest laptop (see annex for its caracteritics).

5 Application to a RANS Navier Stokes solver

5.1 Strategy

The strategy used to compute the fluxes around a cell is an edge based data structure. The code basically goes through the edges that constitute the domain, compute the numerical fluxes and then updates its 2 adjacent cells.

In a serial edge based data structure, the edges are dealt with one at a time, calculating the flux across it, then adding the respective values to one cell and subtracting from the other as necessary. In the situation of figure 16, flux of edges 1 is computed and cells a and b are updated. Then fluxes across edges 2 and 3 will be sequentially computed too.
However, on a parallel code both cells a and cells b will be dealt simultaneously, as will all the cells around them. This means that each cell will have several threads attempting to update it at once (three in the case of the triangular mesh used in this project), causing conflicts within the memory and leading to only one thread successfully updating the value. This is known as a race condition:

![Diagram of race condition]

Figure 17: An illustration of a race condition when running code on parallel environment. On the top, a serial CPU execution: the memory location for the variable of cell a is sequentially accessed. On the bottom, a parallel execution: memory accessing by three threads at the same time.

So in spite of what was illustrated with the matrix/vector multiplication algorithm in previous section, it is not possible to run concurrently on the GPU the computation of all the edges. However this can be overcome by using a colouring scheme, which sorts edges into groups, or “colours”, in such a way that no one cell ever has two edges of the same colour. If these groups of edges are batch processed in parallel, this will eliminate any chance of a race condition, since each cell will only be updated by one edge at a time. Figure 18 illustrates the strategy.
Figure 18: Example of a grid before and after the colouring process. All three edges of one cells are mandatorily of different colours, so that if colours are sequentially computed, two edges cannot access the same cell.

A total of 3 different group of colour will lead to 3 call to the kernel responsible for the computation of the convective fluxes. Each call will processed the edges of the colour in parallel. As each colour don’t have the same number of edges, this require a new set up of the GPU (number of blocks, number of threads per block) at each call.

Figure 19: Sequential execution of each colour.

5.2 Code architecture

In figure 20 on the following page is described the basic structure of the code. During the execution on the GPU, each iteration 3 main routines are in charge of computing the local time step (Time Stepping kernel), the dissipation, and the viscous term of equation (59). Then a 5 step Runge Kutta loop is performed as described in section 6.2.2. Only the convective term of equation (59) is computed at each step, the viscous term and he dissipation remains the same during the 5 stages. Finally the convergence criterion is computed and transferred back to the CPU.
Initialization:
- Read grid file
- Compute normal, cells area...
- Reorder and color edges
- Variable initialization

Data transfer:
- Transfer data from CPU to GPU

GPU computation

Data transfer:
- Transfer convergence criteria “test” to CPU

Test < tol?

Data transfer:
- Transfer data from GPU to CPU for post treatment

Computation of the local time step

Computation of the dissipation term

Computation of the viscous term

Computation of the boundary conditions

Computation of the convective term

Computation of the residual

Smoothing operation

Update of variables

Computation of the convergence criteria

Figure 20: Global structure of the code
Part V

Preliminary results

A fully working GPU RANS version of the code is for the moment still not available. The strategy was to develop first a CPU version of the S-A, in order to then port it on the GPU. By lack of time, due to some problems due to concurrent access to memory during the viscous term calculation, all the code hasn’t been ported yet. However we provide here preliminary results concerning the current CPU OpenMP version of the RANS code, that has been developed before. The code was tested on case 6 from AGARD report [10], an experiment performed by Cook et al.. The grid used to do this computation was the following (and will be discussed later):

<table>
<thead>
<tr>
<th>Grid information</th>
<th>RAE2822</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cells</td>
<td>40960</td>
</tr>
<tr>
<td>Number of edges</td>
<td>61776</td>
</tr>
<tr>
<td>Number of points</td>
<td>20816</td>
</tr>
<tr>
<td>Number cells at wall</td>
<td>224</td>
</tr>
</tbody>
</table>

Table 3: Grid information for test case 6

![Figure 21: RAE2822 grid for RANS calculation](image)

The grid ensure a first cell at a maximum of \( y^+ = 2 \) around all the airfoil. The following table provides the parameters of the simulation. It is to be noted that the angle of attack has
been reduces from half a degree according to [6], due to some inconsistency in the original experiment.

<table>
<thead>
<tr>
<th>Simulation parameters</th>
<th>Agard case 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angle of attack $\alpha$</td>
<td>2.44</td>
</tr>
<tr>
<td>Reynolds number</td>
<td>$6.5 \times 10^{-6}$</td>
</tr>
<tr>
<td>Mach number</td>
<td>0.725</td>
</tr>
</tbody>
</table>

Table 4: Grid information for test case 6

The following graph shows the history of convergence:

![Convergence history graph](image)

Figure 22: Convergence history for test case

The following figure presents comparison to the experiments for the pressure coefficient:
The position of shock is not well predicted by the model. Moreover some slight discrepancies appears around the trailing edge even for the lower surface. On the following page will be found pressure, mach and turbulent viscosity contour. In spite of results close to the physical reality, better results with Spalart Allmaras model was reported in [6] for example. The differences observed here are probably due to the quality of the grid. The number of points in the boundary layer is not sufficient enough to fully solve it. A parallel work of Gaetan Loupy (ENSEEIHT, Toulouse) was supposed to provide a conversion tool to use grids from Gmsh. However, a lack of robustness of the code made it hard to converge with those grids. So this cannot be considered as a proper validation test case, and further validation will have to be made on grids adapted to the S-A model.
(a) Pressure contours

(b) Mach contours

(c) Horizontal velocity contours

(d) Turbulent viscosity contours

Figure 24: Pressure, Mach, Velocity and turbulent viscosity contours
Part VI

Performance and optimisation

As no GPU version of the RANS code is working yet, this optimisation section will only focus on Euler and laminar versions of the code. But all the developments made here, could be easily extended in future work to the RANS formulation.

1 Brut performance

In a previous work, Conor Doherty [11] partially ported a Euler version of the code on GPU. On a 26000 cell mesh of a NACA 0012 the Dissipation and Time-Stepping subroutine (see figure 20) were ported to GPU. The global overall speedup of the code compared to a serial execution shown to be of 1.2. However, in spite of the IEEE 754 norm [32] the code appeared to converge in a different number of iteration than the CPU version. The GPU code converged faster, giving a final speed up of 1.7. We briefly recall to the reader the brut performances of this Euler GPU solver, as it was reported:

Figure 25: Kernelized time-stepping and dissipation subroutine time execution

Considering the performance obtained in literature and the relatively high theoretical power of NVidia Tesla C2050, those results are very poor. Conclusions are that a direct portage of a serial code on a GPU is not enough to take fully advantage of the power of GPU chips. Indeed, due to its particular type of memory and its different architecture, GPU
algorithms required particular treatments in order to accelerate significantly the code. The following sections will present most of them, as they are described by NVidia developers, and the way they had been implemented in our code.

2 Level of optimisation

2.1 Optimisation of algorithms for the GPU

When executing calculations on GPU it is primordial to keep the GPU busy as often as possible. Indeed, the more computation on the GPU the less costly data transfers between CPU and GPU. Even low parallelism computations can sometimes be faster than transferring back and forth to host. This is why only one double precision number is transferred to CPU per iteration: all the operations on data are made on the GPU and kept on the GPU, only the convergence criteria is then transferred at the end of an iteration to CPU to check convergence.

2.2 Optimization of Memory Accesses

GPU global memory is accessed by the threads a lot of time during a calculation. Most of the time during the execution of a kernel, each thread of a block need to read data from the VRAM. On GPU architecture, the time this execution will take is highly sensitive to the physical memory location of those data: this phenomenon is called coalescence. If within a block data accessed by threads are close in memory, one can speak about coalescent memory access. This principle is illustrated on figure 26:

![Coalescent and non-coalescent memory access in the GPU](image)

Figure 26: Illustration of coalescent and non-coalescent memory access in the GPU.
Basically, this means that the more “far” the data are one from another in the memory, the more time will be needed to load them. Non-coalesced memory access is the first source of poor performance of most of the CUDA program and priority should be given on this level of optimization. Coalesced access of the data in memory is however generally not an easy task and sometimes requires shrewd ordering.

With the edge-based strategy and the colouring scheme used to avoid race conditions, memory access is from a first approach intrinsically non-coalescent as illustrated on figure 27:

![Diagram](image)

Figure 27: Illustration of non-coalescent memory access for edge-based strategy on the GPU.

As the thread are randomly attributed to each color, when a color is processed on the GPU, the physical location of the edge can be really different, resulting in reading data from cells that are stored far one from another in the memory. No real effort has in a first time been made to solve this problem.

### 2.3 The advantage of shared and local memory

As mentioned in section 1 each block in a grid can access another type of memory: the shared memory. This memory can only be accessed by threads inside a block and it is of the lifetime of a kernel. This means that all data in shared memory are destroyed at the end of the execution of a kernel. However, advantage of shared memory is that it is hundreds of times faster than global memory. Basically, if redundant non-coalescent access to VRAM are made in a block, those data can be load once in shared memory, then read from this memory to avoid those non-coalescent access. Another advantage, which will be discussed later is that
threads can cooperate via shared memory.

Figure 28: Illustration of loading data in shared memory. The first access is highly non-coalescent, but then all accesses of the shared memory will be faster and coalescent.

Register memory works exactly the same way, except that it can only be accessed from each thread. Following the same principle, data can be loaded once in register memory, then read from this memory to avoid non-coalescent access. However, the amount of register memory is not big, and one must be careful, because data placed in register memory can be transparently put in global memory by the compiler, and harshly slow down the performance of the code.

2.4 Optimization of the parallel environment set up

The set up of the parallel environment is one of the key points of CUDA programming language. Indeed, the partition of the computation should be able to keep the GPU multiprocessors equally busy. A bad set up can lead to very poor performance. Basic rules exist:

- The number of blocks should always be superior to the number of multiprocessors, so that all multiprocessors have always at least one block to execute.

- The number of threads per block should ideally be a multiple of 32, as data are loaded from the memory by wrap of 32.

Ideally, a good set up for our Tesla C2050 would be a high number of blocks to keep the GPU busy, and a number of threads of 32, 64, 128, 256... However, this is not so simple, as the preprocessing step decided how many edges are given to each color. Let illustrate this on a simple triangular grid of 40073 edges. After the preprocessing colouring scheme, the code ends with 7 colours. 5 colours are composed of 6636 edges, 1 with 6637 and the last with 256. The challenge is then to decomposed each colour into a number of block and thread that
is satisfying enough to keep a good occupancy of the GPU. Basically, this is done through a decomposition in prime factor of those numbers:

\[ 6636 = 2^2 \times 3 \times 7 \times 79 = 84 \times 79 \]

84 blocks of 79 threads is probably the combination that will keep the GPU the more busy for the 5 first colors, but is really far from a 100% efficiency. The problem is worth when dealing with the 6637 edges of colour 6. Indeed, this is a prime number and there no other choice can be made other than having 6637 blocks of 1 thread. This harshly slow down the performance of the code in some cases. In this state, the set up used by the code is the following:

<table>
<thead>
<tr>
<th>Colors</th>
<th>N. of edges</th>
<th>N.Blocks x N.Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 5</td>
<td>6636</td>
<td>84 x 79</td>
</tr>
<tr>
<td>6</td>
<td>6637</td>
<td>6637 x 1</td>
</tr>
<tr>
<td>7</td>
<td>256</td>
<td>32 x 8</td>
</tr>
</tbody>
</table>

Table 5: Set up for a 7 color grid with 40073 edges

To illustrate the importance of the set up we took back the matrix/vector algorithms and run a multiplication on a matrix of size \( N = 13440 \) with different set up. The time took to run 10 multiplication is reported in the following histogram:

![Execution time of 10 matrix/vector multiplication, \( N = 13440 \)](chart.png)
The execution time can be multiplied by 2 if the parallel environment is not set up properly. The occupancy of the GPU depends on a lot of different factors, that really needed to be explored. What can be noticed is that the better compromise is found where the number of block is high enough, with a number of thread multiple of 32. For our study we will retain those two main rules.

As we discussed previously, in our CFD code the number of edges per color, a fortiori the number of blocks and threads needed for each of them, are dependent of the coloring scheme and the mesh. The idea to solve this problem is to give each color its edges plus a number of ghost edges that will not affect the computation, but will allow to process this color with an appropriate set up. This process is called padding.

Going back to our previous example of a 40073 edges mesh, considering that we would like 64 threads per block, we will now adopt the following set up:

<table>
<thead>
<tr>
<th>Colors</th>
<th>N. of edges</th>
<th>New N. of edge</th>
<th>Ghost edges</th>
<th>Bl. x Th.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 5</td>
<td>6636</td>
<td>6656</td>
<td>20</td>
<td>104 x 64</td>
</tr>
<tr>
<td>6</td>
<td>6637</td>
<td>6656</td>
<td>19</td>
<td>104 x 64</td>
</tr>
<tr>
<td>7</td>
<td>256</td>
<td>256</td>
<td>0</td>
<td>4 x 64</td>
</tr>
</tbody>
</table>

Table 6: Set up for a 7 color grid with 40073 edges with padding

2.5 Avoiding divergent branches

When a block is executed on a SM, each thread executes the same kernel. Adding conditional if...else statements to make threads doing different calculations may slow down the performance of the code. Indeed, considering our previous problem of set up, one could argue that if a proper set up is used then in the running kernel, if statement could be sufficient to handle the calculations.

For example, if colour 1 is 6636 edges, one solution could be to process this colour on the ideal set up (104 Blocks x 64 threads) and preventing the extra 20 threads to do anything with an if statement. This could avoid the add of the previously discussed 20 ghost edges. However, this strategy highly slows down the performance of the code.

3 Optimized performance

All those strategies were applied to the code. It was then run on a NACA 0012 profile at $M = 0.7$, with an angle of attack $\alpha = 1.0$ on a coarse, medium and fine grid.
Figure 30: 3 types of grids used to compare results
**Coarse grid**

Convergence to $10^{-6}$ was reached in 24362 iterations on both CPU and GPU. Figure 31 draws a detailed time comparison between CPU and GPU for the three main routines of the code. The global acceleration can be found in Table 7:

![Execution time comparison on coarse grid](image1)

![Runge Kutta loop details](image2)

(a) Time comparison CPU/GPU  
(b) Detail of the 5 step RK loop

Figure 31: Time comparison on coarse grid

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Total execution time (s)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>132.1</td>
<td>1.0</td>
</tr>
<tr>
<td>GPU</td>
<td>42.6</td>
<td>3.1</td>
</tr>
</tbody>
</table>

Table 7: Overall speed up on coarse grid

For such a small grid, the speed up is non negligible. **Time stepping and dissipation calculation show a respective speed up of 5.1x and 6.5x**, but the overall speed up is only of 3.1x. This is due to the residual calculation that appears to be a bottleneck during the calculation. It can be seen in figure 32b, that the computation of the boundary conditions (BC) is even 10x slower on the GPU. This is mainly due to the high level of divergent embranchment that can be found in this kernel (a lot of if statements).
**Medium grid**

The results tend to improve when the size of the grid increases. The following results were obtained on the medium grid. Convergence to $10^{-6}$ was reached in 39300 iterations on both CPU and GPU.

![Execution time comparison on medium grid](image)

(a) Time comparison CPU/GPU

![Runge Kutta loop details](image)

(b) Detail of the 5 step RK loop

Figure 32: Time comparison on medium grid

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Total execution time (s)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>493.6</td>
<td>1.0</td>
</tr>
<tr>
<td>GPU</td>
<td>98.5</td>
<td><strong>5.0</strong></td>
</tr>
</tbody>
</table>

Table 8: Overall speed up on medium grid

Time stepping and dissipation calculation show a respective speed up of 9.5x and 8.7x, but the overall speed up is again lower: only 5.0x. The results tend to improve with bigger grids.
**Fine grid**

![Graph](image)

(a) Time comparison CPU/GPU  
(b) Detail of the 5 step RK loop

Figure 33: Time comparison on fine grid

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Total execution time (s)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>1342.5</td>
<td>1.0</td>
</tr>
<tr>
<td>GPU</td>
<td>240.81</td>
<td><strong>5.6</strong></td>
</tr>
</tbody>
</table>

Table 9: Overall speed up on fine grid

Generally speaking the Kernels responsible for the time-stepping and dissipation calculation achieve a respective acceleration of about 9x and 8x. This potential speed up is hindered by the three kernels involved in the 5 step Runge Kutta loop. Indeed In the better case, an acceleration of 3.0x is obtain for this loop. This results in an overall speed up of 5.6x. All the results are summarized in the following table:

<table>
<thead>
<tr>
<th>Grid</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse</td>
<td>3.1</td>
</tr>
<tr>
<td>Medium</td>
<td>5.0</td>
</tr>
<tr>
<td>Fine</td>
<td><strong>5.6</strong></td>
</tr>
</tbody>
</table>

Table 10: Overall speed using GPU
Most of the computer available today come with 4 CPU cores. The results obtained here are slightly better than a 4 core OpenMP version of the code. In this state, the gain of porting a code to GPU is not obvious. Some authors cite acceleration of almost 20, indicating that the approach is far from reaching the theoretical maximum power the GPU can provide.

As described in section 2, we mainly focused on second level optimisations (Environment set up, avoiding branchement, reducing communications...). The main bottleneck resides in the memory treatment of the data. A coalescent access to memory coupled to an efficient use of register/shared memory is the way to really improve the performance of the code. Some authors developed shrewd way to reorder the edges, but it is sometimes really complicated to implement or added some non negligible computational cost. Hence this work proposes a new approach.

4 Domain decomposition based algorithm

4.1 Initial conjecture

In an edge-based (EB) strategy each thread is affected to one edge. Considering a cell $a$, and its neighbours $b$, $c$ and $d$, in order to compute one term of equation (59), data from cell $a$ have to be read 3 times and data from the neighbours once (each edge compute the fluxes according to a JST scheme, so read one time the value of $a$ and one time he value of the neighbour) and then memory is written 3 times (each face added its contribution). Finally we have 9 memory access to compute cell $a$.

If now each thread is affected to one cell, the thread responsible for cell $a$ will only have to read the memory 4 times, and write it 1 time. This defined a cell-based (CB) strategy, as illustrated below:
Let’s now supposed that all threads of a blocks runs calculations on contiguous cells, that is to say that each block represents a contiguous fraction of the entire domain. If the size of this partition satifies to some conditions, each cell could load its own data in shared memory. Then each cells could cooperate into this block.

This is basically the kind of domain decomposition strategy that is udes in MPI CFD codes. It appears to be from a memory point of view particularly well suited for GPU applications. Moreover, a reodering of the cells could lead to more coalescent memory access.

## 4.2 1D compressible Euler equation

In order to define if this idea was indeed capable of improving the velocity of the code, a preliminary 1D Euler code was written in both edge-based (EB) and cell-based(CB) domain decomposition approach. It uses a classical JST scheme, as defined previously and a forward explicit Euler time scheme. A classical Sod shock tube probelem was then run, in order to evaluate the potential of the method. The corresponding GPU code can be found in annex. As no race conditions could appear in 1 dimension, colours were simulated. The details of the simulation are sum up in table 11.

<table>
<thead>
<tr>
<th>Simulation parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the domain</td>
<td>1 m</td>
</tr>
<tr>
<td>Number cells</td>
<td>100</td>
</tr>
<tr>
<td>$\Delta x$</td>
<td>$10^{-2}$ m</td>
</tr>
<tr>
<td>$\Delta t$</td>
<td>$10^{-4}$ s</td>
</tr>
</tbody>
</table>

Table 11: Simulation parameters of Sod’s shock tube problem
with the following initial conditions, for left and right states of an ideal gas:

\[
\begin{pmatrix}
\rho_l \\
\rho_l \\
v_l
\end{pmatrix} = \begin{pmatrix}
1.0 \\
1.0 \\
0.0
\end{pmatrix}, \quad \begin{pmatrix}
\rho_r \\
p_r \\
v_r
\end{pmatrix} = \begin{pmatrix}
0.125 \\
0.1 \\
0.0
\end{pmatrix}
\] (84)

For a validation purpose, the results are given at t=0.2s and compared to theoretical results:

![Density plot](image1)
![Pressure plot](image2)
![Energy plot](image3)
![Velocity plot](image4)

Figure 35: Analytical and numerical solution at t=0.2s for the Sod shock tube problem

The analytical results were providing by a small Fortran code from Dr Timmes from Arizona State University [2]. The results match the analytical solution. In order to get significant time measure, the number of points in the simulation had been increased to $10^4$, and $\Delta t$ adjusted to $10^{-6}$. The following graphs provide the time comparison between the edge-based code and the domain DD version.
This tends to confirm that a domain decomposition algorithms implemented on GPU could lead to good performance, by exploiting better its memory. Next step is the implementation of this strategy for 2 dimensional problems.

### 4.3 2D compressible Euler equations

#### 4.3.1 Metis software

In order to generate a partition for the grid uses in the code, a fast partitioner software has been developed at the Department of Computer Science & Engineering at the University of Minnesota. This software called Metis [18] can be easily integrated into the preprocessing step and take in input the number of partition desired. The program then generate equal partitions and via an output file specify in which partition each cell goes.

![Figure 36: 28 subdomain on a NACA0012 profile comporting 10682 cells](image)

The idea is now to attribute each subdomain to a block, and each cell of this subdomain to a thread in this block. This is why block/subdomain and cell/thread can be confused.
4.3.2 Reordering

In order to take fully advantage of the GPU more coalescent memory acces is desired. As one block of a kernel will be responsible for one domain, and each thread responsible for one cell in this domain, a reordering of the cells is necessary: each cells of a domain have to be contiguous in memory. This will ensure that data read from memory during the execution of a kernel are close one from another.

Figure 37: Illustration of memory location of data relative to each cell before/after reordering process

4.3.3 Halo cells

As a classical MPI code, the cells at boundary of each sub domain may cause problems. Indeed, one or more of the cells required to compute the desired values are located in another sub domain, as illustrated in figure 38. To ensure that this calculation will be performed properly, halo cells are introduced in the considered sub domain. Those cells are ghost cells, in the way that they have no physical meaning, but contain the value of the cells located in other subdomain.
At the end of each iteration, all the halo cells have to be updated in order them to have the updated value from the cell they point to. It is to be noted that boundary conditions are still treated the same way (see section ??)

4.3.4 Padding

Each subdomain have finally to contain the cells of this subdomain plus the halo cells necessary to the computation. The size of those new subdomain may not be rigourously equal. Moreover, we’ve see in section 2.4 that the number of threads in a block should multiple of 32 could give better performance. The idea of a padding was kept to ensure equal number of threads per block and a number of threads per block multiple of 32.
Figure 39: Illustration of padding principle. A padding of 18 is illustrated here: a block of contiguous data in memory contain the cells + halo cells + cells need to fill space up to 18 (padding cells)

512 threads per blocks was chosen in our code. Consequences are that subdomain with less than 512 cells are needed.

4.3.5 Thread cooperation

As described in 2.3, data will be loaded from the global memory into the shared memory by the threads once. All the data needed will then be available directly from the shared memory, making accesses faster and avoiding non coalescent memory access. When two threads try to access the same data in shared memory, bank conflicts can appear. In that case the access to data is serialized by the GPU. This is the major source of slow performance when using shared memory. It can be considered as the equivalent than race condition in EB strategy. However, serialized access to shared memory might still provide better performance than the previous version.

4.3.6 Implementation and validation

The global architecture of the code remain the same than described in figure 20, the strategy behind the calculation of each term has however completely change. Partitioning, reordering, affection of halo cells and padding is part of the preprocessing step. Each routine of figure 20, is then run on one kernel, each block of this kernel running calculations on one subdomain. This only need the add of one subroutine, that will update all the halo cells at the end of each iteration. The code was validated using data from inviscid calculation of Jameson and NASA.
RAE2822 test case 1

<table>
<thead>
<tr>
<th>Grid information</th>
<th>RAE2822</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cells</td>
<td>20015</td>
</tr>
<tr>
<td>Number of edges</td>
<td>30268</td>
</tr>
<tr>
<td>Number of points</td>
<td>10253</td>
</tr>
<tr>
<td>Number cells at wall</td>
<td>325</td>
</tr>
</tbody>
</table>

Table 13: Grid information for validation test cases

(a) Global view  
(b) Detail  

Figure 40: Naca0012 grid

This case is a subsonic case. The following parameters were applied:

<table>
<thead>
<tr>
<th>Simulation parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mach</td>
<td>0.3</td>
</tr>
<tr>
<td>Angle of attack</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 14: Simulation parameters of NASA test case 1

It is interesting because for a shock-free, inviscid flow, there should be no drag generated as the pressure forces acting on the airfoil cancel in the streamwise direction.

Convergence to $10^{-5}$ was reached in 14930 iterations. Following graph shows convergence history:
(a) Convergence history  

(b) Lift and drag coefficient convergence history

Figure 41: Convergence

Mach and pressure contour are given:

(a) Pressure contour  
(b) Mach contour

Figure 42: RAE2822 grid

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>NASA</th>
<th>Our code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_l$  lift coefficient</td>
<td>0.277441</td>
<td>0.276656</td>
</tr>
<tr>
<td>$C_d$  drag coefficient</td>
<td>2.96066x10^{-4}</td>
<td>1.422666x10^{-3}</td>
</tr>
</tbody>
</table>

Table 15: Drag and lift at Mach 0.3, $\alpha = 0.0$

**Jameson test case 2**

This case was performed on a triangular NACA 0012 grid with the following properties:
Grid information

<table>
<thead>
<tr>
<th></th>
<th>NACA0012</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cells</td>
<td>28892</td>
</tr>
<tr>
<td>Number of edges</td>
<td>43634</td>
</tr>
<tr>
<td>Number of points</td>
<td>14742</td>
</tr>
<tr>
<td>Number cells at wall</td>
<td>555</td>
</tr>
</tbody>
</table>

Table 16: Grid information for validation test cases

This case presents the appaition of a shock on both upper and lower surfaces. The following parameters were applied:

<table>
<thead>
<tr>
<th>Simulation parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mach</td>
<td>0.8</td>
</tr>
<tr>
<td>Angle of attack</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Table 17: Simulation parameters of Jameson test case 2

Convergence to $10^{-5}$ was reached in 10930 iterations. Following graph shows convergence history:

Figure 43: NACA0012 grid
Comparison of the pressure coefficient and Mach and pressure contour are given:

Figure 44: Convergence

Figure 45: Pressure coefficient from our code and results from Jameson
The numerical results match those from Jameson. A sharper shock is however obtained, probably due to the way artificial dissipation was tuned. This is more noticeable on the lower surface, were a shock is clearly present in our calculation but is very smooth in Jameson computation. Finally we give here the lift and drag coefficient for this flow:
Coefficient | Jameson | Our code |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_l$ lift coefficient</td>
<td>0.3522</td>
<td>0.3381</td>
</tr>
<tr>
<td>$C_d$ drag coefficient</td>
<td>0.0226</td>
<td>0.0231</td>
</tr>
</tbody>
</table>

Table 18: Drag and lift at Mach 0.8, $\alpha = 1.25$

Those difference are probably due to the trailing edge of our NACA0012 aerofoil.

4.3.7 Performance

The code was run over different grids for a NACA 0012 profile at $M = 0.5$ and $\alpha = 1.0$. The results presented here are time to reach a convergence to $10^{-5}$. All codes used converged in the exact same number of iterations.

Figure 47: Time comparison between the serial, parallel edge based and domain decomposition code
<table>
<thead>
<tr>
<th>N. of cells</th>
<th>N. of partition</th>
<th>N. of halo cells</th>
<th>EB speed up</th>
<th>D.D. speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>12</td>
<td>956</td>
<td>3.5</td>
<td>6.2</td>
</tr>
<tr>
<td>10682</td>
<td>28</td>
<td>2198</td>
<td>4.2</td>
<td>8.3</td>
</tr>
<tr>
<td>23985</td>
<td>62</td>
<td>5069</td>
<td>5.1</td>
<td>9.8</td>
</tr>
<tr>
<td>30195</td>
<td>84</td>
<td>6675</td>
<td>5.3</td>
<td>10.5</td>
</tr>
<tr>
<td>40777</td>
<td>112</td>
<td>9067</td>
<td>5.2</td>
<td>13.7</td>
</tr>
<tr>
<td>83214</td>
<td>250</td>
<td>20560</td>
<td>5.8</td>
<td>14.4</td>
</tr>
</tbody>
</table>

Table 19: Speed up comparison over different grid size

The code shows really good performance especially on large grid, which is particularly encouraging. Next step is to develop a RANS version of this code in parallel to the development of the EB RANS code. Then performance test should be performed to determine which version is the most suitable for GPU applications. Not only in terms of speed up, but in terms of robustness too.
Part VII
Conclusion

The use of GPU in Computational Fluid Dynamics is highly promising. Most of the new supercomputers now include Hybrid architecture with both CPU and GPU which allow considerable accelerations. This study shows that obtaining those accelerations is not as simple as it looks like and some programming efforts are necessary. The code based on an edge-based strategy shows good performance when used on CPU with OpenMP, but this approach is not well-suited for GPU architecture. A complete rethinking of the algorithms are sometimes necessary to take fully advantage of the GPU architecture.

Memory treatment is the key point when dealing with GPU. The way data are stored in memory, the way those data are accesses might seriously hinder the performance of the code. Those aspects have to be explored deeper, with more parametric tests to see the influence on each set up of the parallel environment. The results obtained in this work are nevertheless really encouraging. A speed up of about 15 was obtained on Euler calculation. Which is close to the best accelerations reported in the literature for such double precision codes.

The development of a fully working GPU RANS version of the code is now primordial. The CPU version developed in this work seems to give encouraging results, the GPU version not being far from being finished. The use of a Spalart-Allmaras model seems to be appropriate to GPU applications. However, as a Low-Reynolds model, S-A requires a fine treatment of the viscous boundary layer. This results in particularly fine grid near walls in order to obtain $y^+ = 1$ with enough point to fully solve the boundary layer. This strategy should be discussed in further development. For example, recent development of Steven Allmaras [3] introduces a new wall function, particularly well-suited for S-A model. Good results were reported on transonic flows in a paper from Marsha Berger et al. [6]. Other models, like High-Reynolds $k – \varepsilon$ model have to be analyzed and tested to see if a gain in precision and acceleration is obtained.

Finally, both EB and CB domain decomposition RANS codes should be developed. Indeed, in spite of its really good speed up, communication time to update halo cells in the CB code may seriously hinder the performance when dealing with RANS system of equations. The problem could even get worse when treating 3D problems. Once both version are developed, performance and robustness tests should be run in order to define the most effective strategy.
The CUDA Matrix/Vector multiplication code is given here:

```
#include <stdio.h>
#include <time.h>
#include <systimes.h>
#include <string.h>
#include <malloc.h>
#include <math.h>

__global__ void matmul( float *d_P, float *d_Q, float *d_R, int N );
void matmulCPU( float *P, float *Q, float *R, int N );

int main(int argc, char **args) {
    int T, N;
    T = 16;
    printf("Size of matrix P : ");
    scanf("%d", &N);
    clock_t start, end;

    int i, j;

    P = (float*)malloc(N*N*sizeof(float));
    Q = (float*)malloc(N*sizeof(float));
    R = (float*)malloc(N*sizeof(float));

    cudaMemcpy((void **)&d_P, N*N*sizeof(float));
    cudaMemcpy((void **)&d_Q, N*sizeof(float));
    cudaMemcpy((void **)&d_R, N*sizeof(float));

    for(i=0; i<N; i++) {
        for(j=0; j<N; j++) {
            ...
        }
    }
```

\[ P[N*i+j] = \frac{2.0}{(float)(i+j+1)}; \]

\[ Q[i] = (float)i; \]

\[
\text{cudaMemcpy} (d_P, P, N*N*sizeof(float), cudaMemcpyHostToDevice);
\text{cudaMemcpy} (d_Q, Q, N *sizeof(float), cudaMemcpyHostToDevice);
\]

\[
dim3 M_b(NT,1);
dim3 M_t(T,1,1);
i=0;
start = clock();
while(i<10){
    \text{matmul}<<<M_b,M_t>>>(d_P,d_Q,d_R,N);
    i++;
}
\text{cudaThreadSynchronize}();
end = clock();
\text{cudaMemcpy} (R, d_R, N *sizeof(float), cudaMemcpyDeviceToHost);
\text{fprintf}(stdout,\"GPU in \%f s\n\",(float)(end-start)/CLOCKS_PER_SEC);
\]

\[
i=0;
start = clock();
while(i<10){
    \text{matmulCPU}(P,Q,R,N);
    i++;
}
end = clock();
\text{fprintf}(stdout,\"CPU in \%f s\n\",(float)(end-start))(end-start)/CLOCKS_PER_SEC);
\]

\[
\text{\_global\_ void matmul( float *d_P, float *d_Q, float *d_R, int N ) {}\}
\]

\[
int i = threadIdx.x + (blockIdx.x * blockDim.x);
int k;
\]
float elem = 0.0;
for(k=0;k<N;k++) {
    elem += d_P[i*N+k]*d_Q[k] ;
}
 d_R[i] = elem ;} 

void matmulCPU( float *P, float *Q, float *R, int N ) {
    int i,k;
    float elem ;
    for(i=0;i<N;i++) {
        elem = 0.0;
        for(k=0;k<N;k++) {
            elem += P[i*N+k]*Q[k] ;
        }
    R[i] = elem ;
    }
}
References


