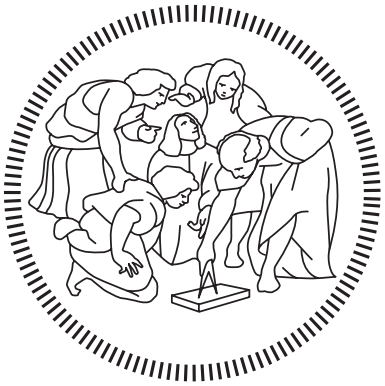
Politecnico di Milano
School of Industrial and Information Engineering
Master of Science in Aeronautical Engineering



A GPU Accelerated Solver for DNS of Homogeneous Isotropic Turbulence

Advisor: Prof. Maurizio Quadrio
Co-advisor: Prof. Aldo Frezzotti

M. Sc. dissertation of:
Daniele Menni
816940

Academic Year 2015-2016

ai compagni di viaggio...

Abstract

A pseudo-spectral code for the direct numerical simulation of incompressible Navier-Stokes equations for homogeneous isotropic turbulence is presented. The program is developed to run on graphical processing units (GPUs), exploiting the massive parallelism of modern GPU architectures. The pseudo-spectral method is based on Fourier transforms: the code works in wavenumbers domain, but non linearities are handled in physical domain instead of executing expensive complex convolution computations. The GPU can handle thousands of concurrent threads since it is composed of hundreds of cores, opposed to a low number of threads concurrently running on a CPU. This characteristic makes operating on large data like matrices very efficient, providing a substantial speed-up compared to traditional CPU parallel code.

Keywords: direct numerical simulation; DNS; graphical processing unit; GPU; CUDA; homogeneous isotropic turbulence; massive parallelism; many-core processor

Sommario

Questo lavoro presenta un codice pseudo-spettrale per la simulazione numerica diretta delle equazioni di Navier-Stokes applicate ad una turbolenza omogenea isotropa. Il programma è sviluppato per essere eseguito sul processore grafico (GPU) sfruttando la natura altamente parallela dei più recenti prodotti di questo tipo. Il metodo pseudo-spettrale si basa sulla trasformata di Fourier: il programma lavora nello spazio dei numeri d'onda ma le non linearità sono trattate nello spazio fisico invece di effettuare convoluzioni nello spazio complesso. La GPU può gestire migliaia di processi simultaneamente essendo composta da centinaia di processori, contrariamente al ridotto numero di processi gestibili contemporaneamente da una CPU. Questa caratteristica permette di operare su grandi strutture dati come matrici in modo molto efficiente, ottenendo una signficativa accelerazione rispetto a un codice parallelo classico per CPU.

Parole chiave: simulazione numerica diretta; DNS; processore grafico; GPU; CUDA; turbolenza omogenea isotropa; parallelismo massivo

Ringraziamenti

Per prima cosa ringrazio la mia famiglia per il sostegno che mi ha dato durante il percorso universitario, soprattutto nei momenti di difficoltà, ciò mi ha consentito di vivere questi cinque anni con meno pressione e più spensieratezza.

Ringrazio poi gli amici dell'università, non semplici compagni di corso ma persone che mi ritengo fortunato di aver conosciuto. Tra questi soprattutto Andrea, Daniele, Francesco & Francesco e Andrea; sono stati un'ottima compagnia per riempire i tempi (spesso eccessivi) delle trasferte in treno quotidiane e non solo.

Un ringraziamento speciale a Gianluca e Mario per essere amici su cui è sempre possibile contare.

Ringrazio inoltre il Prof. Quadrio per il sostegno durante la tesi e il Prof. Frezzotti per aver dato l'input per realizzare questo lavoro. Ringrazio anche il Prof. Guardone per avermi permesso di sfruttare l'hardware a sua disposizione e il Dr. Gori per avermi fatto da supporto tecnico e sopportato le mie numerose e-mail.

Ringrazio infine il Politecnico di Milano per avermi dato la possibilità di passare anni fantastici che, tra alti e bassi, sono letteralmente volati: faccio ancora fatica a credere di essere stato una matricola ben cinque anni fa.

Contents

List of Figures

List of Tables

1. Introduction

Scientific research has always pushed the boundaries of computational resources. Since the invention of the transistor in 1947, the design and manufacturing progresses allowed the production of smaller, cheaper, more efficient and, above all, more powerful computational processors: the objective is to model increasingly complex system and phenomena. This is even more important in the case of fluid flow simulations, where a large number of grid points is needed to produce accurate modes, therefore needing large and expensive computer systems. Real-world flows are characterised by high Reynolds number and require particularly dense grids in order to solve turbulent flows.

Traditionally, computer software has been written for serial computation, as a serial stream of execution of one or more algorithms, to be executed from a central processing unit (CPU) on a single computer.

In the mid 1980's, a new way of building supercomputers was born. Instead of developing a massive proprietary computer, the idea was to use many mass market, off the shelf processors, connected by off the shelf network in an environment called *cluster*. Today, clusters are the backbone of scientific computing and the dominant architecture in data centres.

Until the mid 2000's, computing power has been improved by increasing the working frequency of processors (*clock*), up to a point when the power requirements became too high to allow a frequency increase and Moore's law (an empirical observation that the number of transistors in a microprocessor doubles every 18 to 24 months) wouldn't have been valid. Power consumption scales with processor area, while processor speed scales with the square root of the area. The solution has been *multi-core processors*, which are now common inside consumer products (laptops, desktops, smartphones), and are a more efficient way to increase computational power compared to frequency increase; the transistor scaling predicted from Moore's law also allows to transition from few cores to many cores. More cores inside a processor increase performance through parallelism while minimising power consumption, avoiding excessive processor area increases.

Graphical processing units (GPUs) were initially developed for graphical applications and were programmed with particular graphics languages. Exploiting their power for computational purposed required graphical programming knowledge and disguising data as pixels or textures. This was particularly cumbersome and discouraged developers from learning and using this powerful hardware.

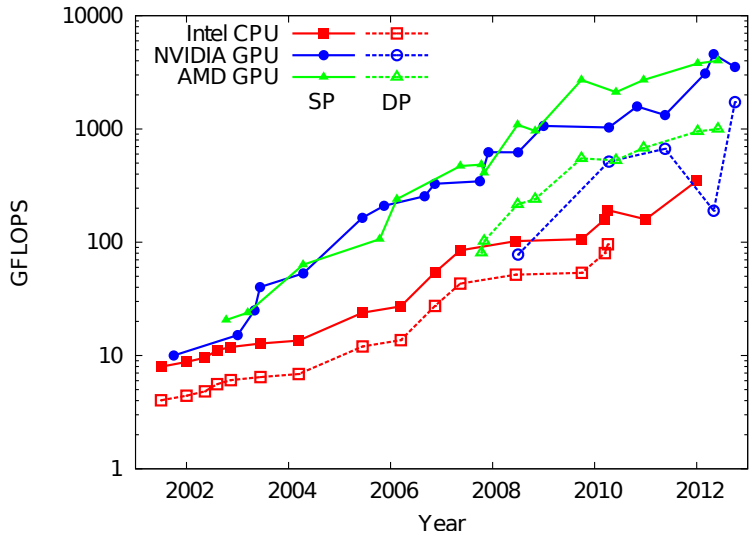


Figure 1.1: Theoretical peak performance evolution in time for Intel CPUs and AMD/Nvidia GPUs, in Gflops (billion floating-point operations per second); SP=single precision, DP=double precision ([NS14]).

As evident from Fig. 1.1, the evolution in theoretical peak floating-point operations per second (flops) for multi-core processors and graphical processing units shows that modern GPUs offer more than an order of magnitude higher performance compared to modern CPUs.

Clearly, this performance advantage could no longer be ignored by the scientific community.

In 2007, Nvidia introduced a the CUDA programming language and toolkit to allow programmers to harness a kind computing power which was only available to graphic applications (and required graphic-programming knowledge to be used for scientific computing purposes), turning them in very powerful co-processors. In 2009, the Kronos Group release a similar but open-source solution called OpenCL, effectively extending the support of this technology to all GPU manufacturers.

These solutions have allowed developers to deploy the calculations on a much higher number of computing cores (hundreds or thousands), since CPUs and GPUs have different architecture, and obtain a substantial acceleration of execution times.

As well documented in [NS14], computational fluid dynamic (CFD) is a field very hungry for computing power, and is always thriving for the latest and greatest technologies in both hardware and software. From the software point of view, some of the more established solutions (like ANSYS Fluent and OpenFOAM) already offer support to GPU acceleration in their

simulation suite.

The inspiration for this thesis came from a demo included in the CUDA toolkit called FluidsGL [Goo12], inspired by the work of Jos Stam [Sta99]. While not interesting or accurate from a scientific point of view (Stam's work has a computer graphic purpose), it sparkled the idea of implementing a Navier-Stokes solver in the CUDA C programming language.

There are three very computationally intensive families of numerical simulations that can be employed to solve the Navier-Stokes equations: Reynolds-averaged Navier-Stokes equations (RANS), large eddy simulations (LES) and direct numerical simulations (DNS), listed from the less computationally intensive and less accurate to the more computationally intensive and more accurate. DNSs in particular require massive clusters of computers to achieve even moderate Reynolds numbers, as detailed in Chapter 4.

This thesis revolves around the implementation of a DNS solver, since it is a perfect candidate for the translation from CPU code to GPU code. A DNS simulation is the best way to study low-Reynolds turbulence since it provides all the statistical quantities, and not just the mean quantities like RANS for example.

This work contains a program that compute the DNS of homogeneous isotropic turbulence. Turbulence is a very complex phenomenon, and this flow is the simplest system in which is possible to observe turbulent phenomena and it is a good foundation to analyse more complex systems like the flow in a pipe. The hypotheses used with homogeneous isotropic turbulence are also true for small scales far from the wall in other type of flows. Moreover, this case is a good choice for parallelisation thanks to the independence of data, a requirement for efficient parallel code, and the need for fast and efficient Fourier transforms, for which the CUDA toolkit provide a highly optimised library to speed up the computation.

There has been some implementations ([LQ06]) of low-cost solutions for this problem in the form of commodity cluster computing, that is the use of a number of already available, affordable and easy to obtain computer hardware for parallel computing, in order to obtain the greatest amount of useful computation at low cost.

The use of a GPU should provide a noticeable speed up compared to classic CPU code, and allow the realisation of a 3D DNS solver able to provide results at a reasonable Reynolds number on a single desktop computer, thanks to the massive parallelism that characterise modern GPU architectures.

This dissertation starts with a description of the tools that made this thesis possible, GPUs, detailing the differences from CPUs, the advantages and disadvantages of this choice, with a description of the toolkit used to manage the accelerator device. The material contained in Chapter 2 is adapted

from the excellent book on the CUDA programming language "Professional CUDA C Programming" by Cheng, Grossman and McKercher (2014); it is a good starting point that I would suggest to anyone willing to learn this programming language and with a bit of ANSI C background.

Chapter 3 contains a small amount of theoretical background on turbulence, followed by Chapter 4 which contains the description of numerical simulations for Navier-Stokes equations and the limitations of the approach used in this work; the content of these chapters is adapted from the famous "Turbulent Flows" by Pope (2000).

Also in Chapter 4 there is the mathematical discussion of the method used in this thesis, based on Fourier transforms to work both in physical and wavenumbers domain.

In Chapter 5 the developed program is explained in its principal parts with explanations of the choices made during development to extract the best performance from the GPUs.

Chapter 6 contains methods and data for the validation process for the program and analysis of the performance gains obtained through this work.

2. The hardware choice

2.1 Heterogeneous parallel computing

Graphical Processing Units (GPUs) evolved greatly in recent times. Originally designed to produce images for output on a display, they are now a serious choice when fast processing is require.

In the past few years, GPUs have been attached to Central Processing Units (CPUs) to accelerate a broad array of computations, realising what is now called *heterogeneous computing*. Today, GPUs are used on desktop systems, computer clusters and even on many of the fastest supercomputers of the world (like the Titan Cray XK7 with 18688 Nvidia Tesla K20X GPUs, second place in the TOP500 ranking). Providing large amount of compute power for technical computing, GPUs have enabled advances in science and engineering in a broad variety of disciplines thanks to a huge number of compute cores that works in parallel while keeping a reasonable power budget.

In the past, to take advantage of GPUs computing power outside the narrow range of applications they were intended for required major efforts, and graphical programming knowledge was a requisite for the GPU programmer.

In 2007 Nvidia released *CUDA*, a much more convenient way to use GPUs for computing purposes. CUDA is one of the most popular application program interface (API) for accelerating a range of compute kernels on the GPU; it enables code written in C, C++, Fortran, and Python to run efficiently on a GPU with reasonable programming effort. It has a good balance between the need to know the architecture in order to exploit it well, and the need to present a readable and easy to use programming interface.

The major difference between parallel programming in C and parallel programming in CUDA C is that CUDA architectural features, such as memory and execution models, are directly exposed to the programmer, thus enabling more control over the hardware.

Parallel computing

From a calculation perspective, *parallel computing* can be defined as a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided in smaller ones and then solved at the same time. Parallel computing can be defined as the simultaneous use of multiple computing resources (cores or computers) to perform calculations. A large problem is broken down into smaller ones, each then solved concurrently on different computing resources.

Parallel computing usually involves two distinct areas of computing technologies: computer architecture (hardware aspect) and parallel programming (software aspect). *Computer architecture* focuses on supporting parallelism at the architectural level, while *parallel programming* focuses on solving a problem concurrently by fully using the computational power of the computer architecture.

Most modern processors implement the *Harvard architecture*, which includes three main components:

- memory: instruction memory and data memory;
- CPU: control unit and arithmetic logic unit (ALU);
- input/output interfaces.

The key component is the CPU, usually called the *core*. Nowadays, the trend in chip design is to integrate multiple cores onto a single processor (multicore) to support parallelism at the architecture level.

When implementing algorithms for multi-core machines, it's very important to be aware of the characteristics of the underlying computer architecture.

Essentially, a program consists of two basic components: instructions and data. When a computational problem is broken down into many small pieces of computation, each piece is called a *task*. In a task, individual instructions consume inputs, apply a function, and produce outputs. A *data dependency* occurs when an instruction consumes data produced by a preceding instruction. Analysing data dependencies is fundamental in implementing parallel algorithms because dependencies are one of the primary inhibitors to parallelism, and understanding them is necessary to obtain application speedup in the modern programming world.

There are two fundamental types of parallelism in applications: task and data parallelism. *Task parallelism* happens when there are many tasks or functions that can be executed independently and overall in parallel; its focus is on distributing functions across multiple cores. *Data parallelism* occurs when there are many data items that can be operated on at the same time; its focus is on distributing data across multiple cores.

CUDA programming is especially well-suited to address problems that can be expressed as data parallel computations. Many applications that process large data sets can use a data-parallel model to speed up the computations. Data-parallel processing maps data elements to parallel threads.

2.2 Computer architecture

One widely used way to classify computer architectures is *Flynn's Taxonomy*, which classifies architectures into four different types according to how instructions and data flow through cores (Fig. 2.1).

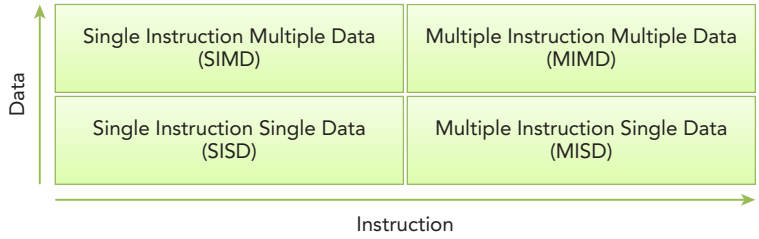


Figure 2.1: Flynn's Taxonomy: classification of computer architectures. [CGM14]

Single Instruction Single Data refers to the traditional computer: a serial architecture, with only one core in the computer. At any time only one instruction stream is executed, and operations are performed on one data stream.

Single Instruction Multiple Data refers to a type of parallel architecture; there are multiple cores in the computer. All cores execute the same instruction stream at any time, each operating on different data streams. Most modern computers employ a SIMD architecture.

Multiple Instruction Single Data refers to an uncommon architecture, where each core operates on the same data stream via separate instruction streams.

Multiple Instruction Multiple Data refers to a type of parallel architecture in which multiple cores operate on multiple data streams, each executing independent instructions; many MIMD architectures also include SIMD execution sub-components.

At the architectural level, the objectives are: decrease latency, increase bandwidth and throughput. *Latency* is the time necessary for an operation to start and complete. *Bandwidth* is the amount of data that can be processed per unit of time. *Throughput* is the amount of operations that can be processed per unit of time, expressed in *gigaflops* (billion floating-point operations per second).

Computer architectures can also be classified by the memory organisation:

- multi-node with distributed memory
- multiprocessors with shared memory

Multi-node systems include many processors connected by a network, each processor has his own local memory, and they communicate the content of their local memory over the network. These systems are typically referred to as *clusters*.

Multiprocessors can vary in size from dual-processors to hundreds of processors; they are either physically connected to the same memory, or share a low-latency link (like PCI-Express). The term *multi-core* is used when tens or hundreds of cores are involved.

GPUs represent a many-core architecture, and have virtually every type of parallelism previously described: instruction-level parallelism, multi-threading, SIMD, and MIMD. Nvidia coined the phrase *Single Instruction Multiple Thread* (SIMT) for this type of architecture. SIMT execution is an implementation choice: sharing control logic leaves more space for ALUs.

2.3 Heterogeneous computing

Homogeneous computing uses one or more processors of the same architecture to execute an application. *Heterogeneous computing* instead employs a variety of processor architecture to execute an application, assigning tasks to architectures which are well-suited and thus yielding better performances.

The effective use of heterogeneous systems is currently limited by the application design complexity, adding another layer of programming effort over traditional parallel programming.

Currently, a GPU is not a standalone platform but a co-processor to a CPU, and must operate in conjunction with it through a PCI-Express bus, as shown in Fig. 2.2. For this reason, the CPU is called the *host* and the GPU is called the *device*.

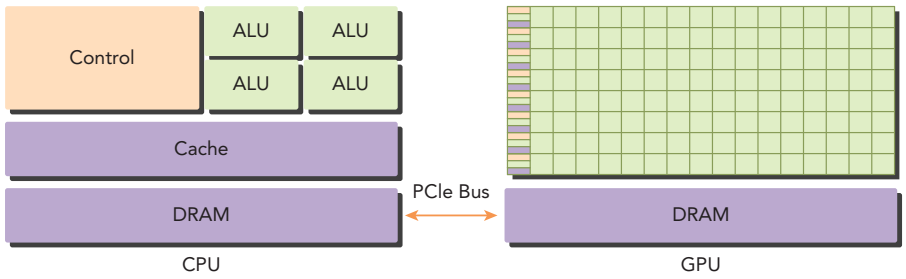


Figure 2.2: Representation of a heterogeneous system.[CGM14]

An heterogeneous application consists of two parts: host code and device code. *Host code* runs on CPUs and *Device code* runs on GPUs. An application running on a heterogeneous computer is initialised by the CPU, since host code is responsible for managing the environment, code and data

for the device before compute-intensive tasks are loaded on it. For computational intensive applications, GPUs are ideal since they accelerate program sections that exhibit a rich amount of data parallelism.

CUDA is enabled on a variety of product families in the Nvidia GPU line-up; the Tesla product family is the most powerful and is oriented to datacenter parallel computing.

There are two important features that describe GPU capability: number of CUDA cores and memory size. There also two different metrics for describing GPU performance: peak computational performance and memory bandwidth. *Peak computational performance* is a measure of compute capability, defines as how many single or double-precision floating point calculations can be processed per second. *Memory bandwidth* is a measure of the ratio at which data can be read from or stored to memory.

GPU computing is not meant to replace CPU computing, since each has advantages for certain kind of situations. CPUs are good control-intensive tasks, they are optimised for dynamic workloads marked by short sequences of computational operations and unpredictable control flow; GPUs are good for data-parallel computation-intensive tasks, and are optimised for workloads dominated by computational tasks with simple control flow.

As shown in Fig. 2.3 There are two dimensions that differentiate the scope of applications for CPU and GPU: parallelism level and data size.

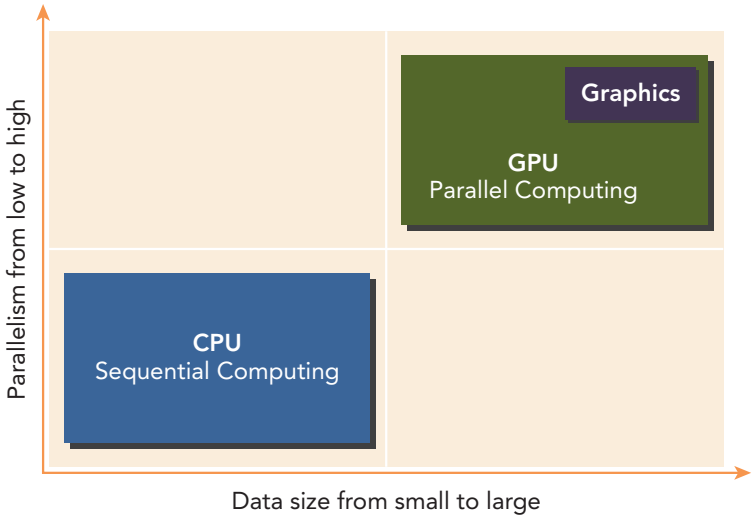


Figure 2.3: Optimal scope of applications.[CGM14]

The difference is more clear if examined at thread level. Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off CPU execution channels to provide multi-threading capa-

bility. Context switches are slow and expensive. Instead, threads on GPUs are extremely lightweight; in a typical system, thousands of threads are queued up for work, and if the GPU must wait on one group of threads, it begins executing work on another.

CPU cores are designed to minimise latency for one or two threads at a time, whereas GPU cores are designed to handle a large number of concurrent, lightweight threads in order to maximise throughput. Today, a CPU with a quad core processors can run only 4 threads concurrently (8 if the CPUs support hyper-threading). Modern Nvidia GPUs can support up to 1,536 active threads concurrently per multiprocessor. On GPUs with 16 multiprocessors, this leads to more than 24,000 concurrently active threads.

2.4 The CUDA programming model

CUDA is a general-purpose parallel computing platform and programming model that leverages the parallel compute engine in Nvidia GPUs to solve computational problems in a more efficient way. It's accessible through CUDA-accelerated libraries (like CUFFT), compiler directives, APIs and extension to various programming languages.

CUDA C, used in this thesis, is an extension of standard ANSI C with several language extensions to enable heterogeneous programming, and also APIs to manage the GPU. It is also a scalable programming model that allows to scale the parallelism to GPUs with varying number of cores.

CUDA provides two API levels for managing the GPU and organising threads: the CUDA driver API and the CUDA runtime API. The *driver API* is a low-level API and is harder to program but it enables more control to the GPU. The *runtime API* is a higher-level API implemented on top of the driver API, and each of its functions is broken down into more basic operations issued to the driver API. There is no noticeable performance difference between them.

Nvidia's CUDA `nvcc` compiler separates host and device code during the compilation process. The host code is compiled with standard C compilers (like `gcc`). The device code is written using CUDA C extended with certain keywords that labels data-parallel functions, called *kernels*; it is then compiled by `nvcc`. During the linking stage, CUDA runtime libraries are added for kernel procedure calls and explicit device manipulation.

Programming models present an abstraction of computer architectures that operate as a bridge between an application and its implementation on the hardware. In addition to this, the CUDA programming model provides special features to harness the computing power of GPU architectures:

- a way to organise threads on the GPU through a hierarchy structure

- a way to access memory on the GPU through a hierarchy structure

A key component of the CUDA programming model is the kernel, that is the code that runs on the GPU. The developer needs to express a kernel as a sequential program. Behind the scenes, CUDA manages scheduling programmer-written kernels on CPU threads. From the host, the programmer defines how the kernel is mapped to the device based on application data and GPU capability. The CUDA programming model is primarily asynchronous so that after a kernel launch, control is returned to the host, leaving the CPU free to perform additional tasks, overlapping computation with the GPU.

A typical CUDA program follows this pattern of major operations:

1. copy data from CPU memory to GPU memory;

2. invoke kernels to operate on data stored in GPU memory;

3. copy back data from GPU memory to CPU memory.

2.4.1 Managing memory

The CUDA runtime provides functions to allocate and release device memory and to transfer data between host and device memory; the functions performing these operations are similar to the ones in ANSI C, like `cudaMalloc`, `cudaMemcpy` or `cudaFree`.

Many workloads are limited by the speed of data load and store, thus it would be ideal to have a large amount of low-latency, high-bandwidth memory, but it's not always economical or possible so the memory model needs to provide optimal latency and bandwidth given the available hardware.

Applications generally follow the *principle of locality*, which means that they access a relatively small and localised portion of their address space at any time. There are two types of locality: temporal and spatial locality. *Temporal locality* assumes that if a data is referenced, then it's more likely to be referenced within a short period and less likely as more time passes. *Spatial locality* assumes that if a memory location is referenced, nearby locations are also likely to be referenced.

Modern computers uses a *memory hierarchy* of progressively lower-latency but lower-capacity memories to optimise performance: this is useful thanks to the principle of locality. A memory hierarchy consists of multiple levels of memory with different latencies, bandwidths, and capacities, as illustrated in Fig. 2.4; at the bottom there are types of storage characterised by lower cost, higher capacity and latency and less frequent access by the processor.

CPUs and GPUs use similar principle and models in memory hierarchy design, the difference is that the CUDA programming model exposes more of it to allow more explicit control over its behaviour.

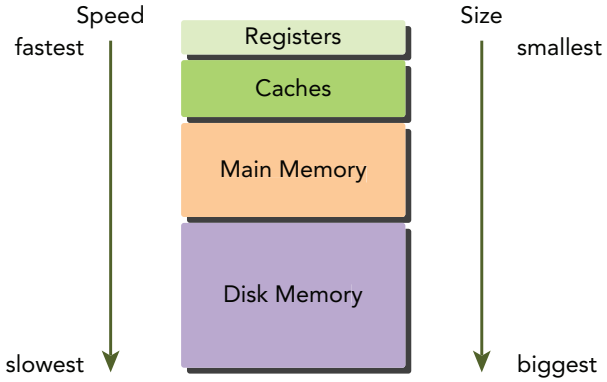


Figure 2.4: Typical memory hierarchy.[CGM14]

For programmers, memory can be classified between programmable and non-programmable, based on the possibility to control data placement in it. The CUDA memory model exposes many types of programmable memory, and Fig. 2.5 illustrates the hierarchy of these memory spaces: each has a different scope, lifetime and caching behaviour.

More specifically, the memory model consists of:

- *registers*: fastest memory space on a GPU, private to each thread, same lifetime of the kernel. They are a scarce resource, partitioned among active warps in an SM; an excessive use of registers results in *register spilling* over to local memory, degrading performance;
- *local memory*: used when registers are full and to store large arrays, it has the same scope and lifetime of registers. Despite the name, it's similar to global memory and has high latency and low bandwidth;
- *shared memory*: on-chip, much higher bandwidth and much lower latency than global memory, same lifetime of a block and scope of all threads in a block. Each SM has a limited amount of shared memory partitioned among blocks, and it is also a mean for inter-thread communication by allowing data sharing between threads within a block;
- *constant memory*: read only, it resides in device memory and is cached in a dedicated, per-SM constant cache; it's only 64KB and has global scope, visible to all kernels;
- *texture memory*: part of global memory, it resides in device memory and is cached in a per-SM, read-only cache. It includes support for hardware filtering and is optimised for 2D spatial locality;

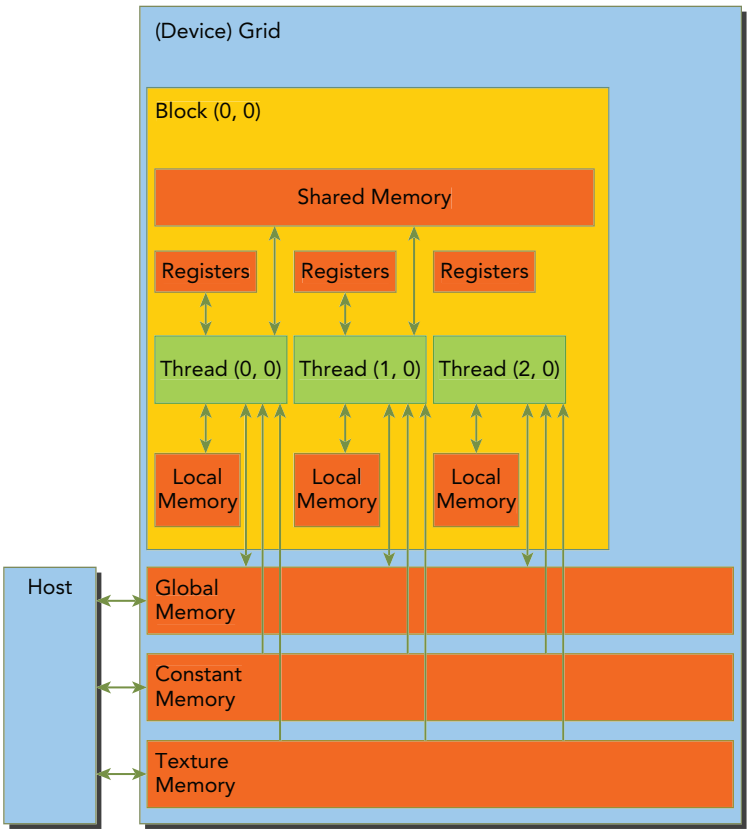


Figure 2.5: CUDA memory model.[CGM14]

- *global memory*: it's the largest, highest-latency and most used memory on a GPU; it resides in device memory and has global scope and lifetime. It's fundamental to optimise memory transactions for global memory to obtain optimal performance.

A GPU also includes four types of non-programmable caches: L1, L2, read-only constant and read-only texture. There is one L1 cache per-SM and one L2 cache shared by all SMs. Both L1 and L2 caches are used to store data in local and global memory, including register spills. On the GPU only memory loads can be cached; memory store operations can't be cached (unlike on the CPU). Each SM also has a read-only constant and read-only texture cache that are used to improve read performance from their respective memory spaces in device memory.

Global memory accesses are staged through caches. All application data initially resides in VRAM (video random access memory), the physical device memory. All accesses go through the L2 cache, and many also pass through the L1 cache.

To improve performance in accessing device memory it's fundamental to

produce aligned and coalesced memory accesses. *Aligned memory accesses* occur when the first address of a device memory transaction is an even multiple of the cache granularity being used to service the transaction (32 bytes for L2 cache or 128 bytes for L1 cache). *Coalesced memory accesses* occur when all 32 threads in a warp access a contiguous piece of memory. Exploiting these characteristics of device memory accesses allows to obtain maximum global memory throughput.

2.4.2 Organising threads

When a kernel function is launched from the host, execution is moved to a device where a large number of threads are generated and each thread executes the statements specified by the kernel function.

CUDA exposes a thread hierarchy abstraction to allow the programmer to organise the threads, since this is a critical part of CUDA programming.

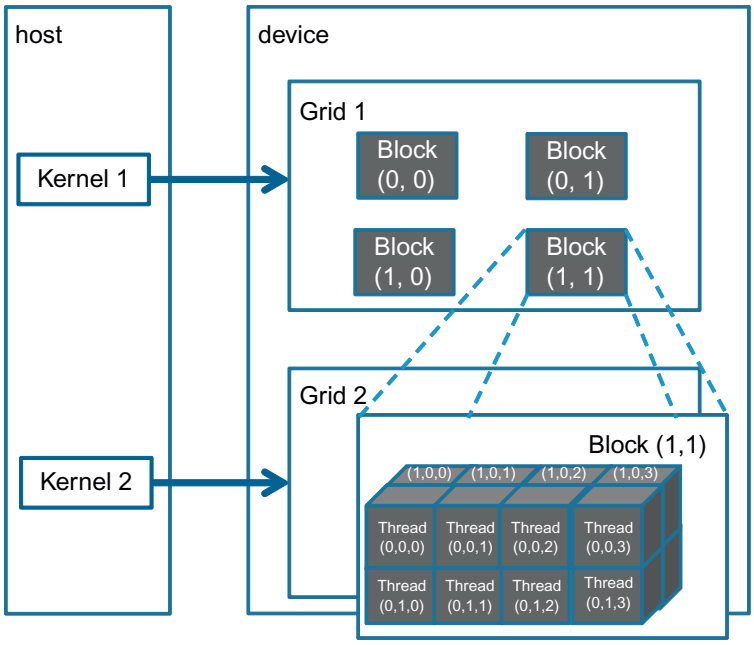


Figure 2.6: Threads hierarchy on the GPU; example with a 2D grid of 3D blocks.[KH10]

As illustrated in Fig. 2.6, the thread hierarchy has two levels and is decomposed into blocks of threads and grids of blocks.

All threads created by a single kernel launch are collectively called a *grid*. All threads in a grid share the same global memory space. A grid is composed of many blocks. A thread block is a group of threads that can cooperate with each other using: block-local synchronisation and block-local shared memory. Threads from different blocks can't cooperate.

To distinguish among the high number of threads, two unique coordinates are provided:

- `blockIdx`: block index within a grid;
- `threadIdx`: thread index within a block.

They appear as built-in, pre-initialised variables that can be accessed within kernel functions. When a kernel function is executed, they are are assigned to each thread by the CUDA runtime. They are of type `uint3`, a CUDA built-in vector type, it's a structure containing three unsigned integers representing the three directions, accessible through the fields `.x`, `.y` and `.z`.

CUDA organises grids and blocks in three dimensions, specified by two built-in variables:

- `blockDim`: block dimension, measured in number of threads;
- `gridDim`: grid dimension, measured in number of blocks.

They are of type `dim3`, similar to `uint3`.

Grid and block dimensionality of a kernel launch affect performance: this abstraction is an additional way to optimise the program. However, there are several restrictions on the dimensions of grids and blocks; one of the major limiting factors on block size is the available resources like registers, shared memory and others.

A CUDA kernel call is ax extension of the standard C function syntax that adds the kernel's *execution configuration* inside triple angle brackets:

```
kernel_name<<< grid, block, sharedSize, stream >>>(argument list);
```

where:

- `grid`: is of type `dim3` and specifies the dimension and size of the grid, such that `grid.x*grid.y*grid.z` is the number of blocks being launched;
- `block`: is of type `dim3` and specifies the dimension and size of each block, such that `block.x*block.y*block.z` is the number of threads per block;
- `sharedSize`: is of type `size_t` and specifies the number of bytes of shared memory that it dynamically allocated per block for this call in addition to the statically allocated memory; it's an optional argument which defaults to zero;
- `stream`: is of type `cudaStream_t` and specifies the associated stream; it's an optional argument which defaults to zero;

A kernel call will fail if `grid` or `block` are greater than the maximum sizes allowed for the device as specified by its *compute capabilities*, or if `sharedSize` is greater than the maximum amount of shared memory available on the device, minus the amount of shared memory required for static allocation.

By specifying the grid and block dimensions it's possible to configure the total number of threads for a kernel and their layout.

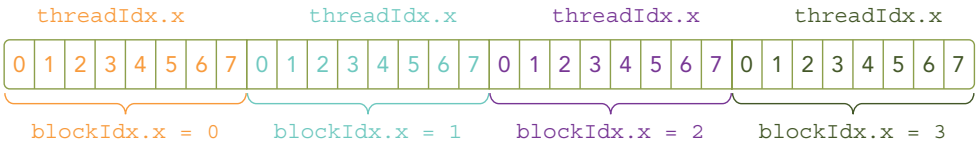


Figure 2.7: Threads layout for a kernel launch with 4 blocks, each with 8 elements each.[CGM14]

Since the data is stored linearly in global memory, the built in variables blockIdx.x and threadIdx.x used in Fig. 2.7 can be used to identify a unique thread in the grid and to establish a mapping between threads and data elements.

A kernel is defined using the `__global__` declaration specification:

```
__global__ void kernel_name(argument list);
```

and must have a `void` return type.

2.5 GPU architechture

The GPU architecture is built around a scalable array of *Streaming Multiprocessors*(SM).

Fig. 2.8 illustrates the key components of a Fermi generation SM:

- CUDA cores;
- shared memory/L1 cache;
- register file;
- load/store Units;
- special function units;
- warp scheduler.

Each SM in a GPU is designed to support concurrent execution of hundreds of threads, and there are multiple of them in a GPU. When a kernel grid is launched, the blocks of that grid are distributed among available

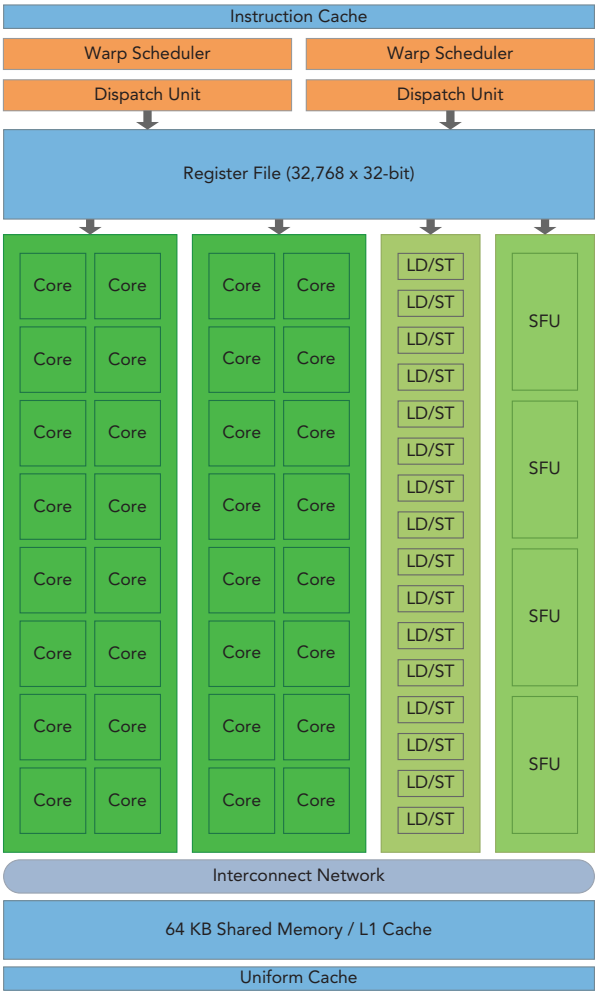


Figure 2.8: Key components of a Fermi GPU.[CGM14]

SMs for execution. Once placed on an SM, the threads of a block execute concurrently only on that assigned SM; it remains there until the execution completes. If multiple blocks are assigned to the same SM at once, they are scheduled based on the availability of the SM resources. Instructions within a single thread are pipelined to leverage instruction level-parallelism, in addition to thread-level parallelism.

Shared memory and registers are important but scarce resources in an SM. Shared memory is partitioned among blocks resident on the SM, registers are partitioned among threads. Threads in the same block can cooperate and communicate with each other though these resources. While all threads in a block run logically in parallel, not all threads can execute physically at the same time, and as a result different threads in a block may make

progress at a different pace.

Because shared memory and registers are limited resources, they impose a strict restriction on the number of active warps in an SM, which corresponds to the amount of parallelism possible in an SM, that is the number of blocks.

2.6 The CUDA execution model

Warp execution

Warps are the basic unit of execution in a shared multiprocessor. A grid of thread blocks is distributed among SMs when a kernel is launched; when a thread block is scheduled to run on a SM, threads in the block are further partitioned in warps. A warp consists of 32 consecutive threads, and all threads in a warp are executed in Single Instruction Multiple Thread (SIMT) fashion: all threads execute the same instruction, and each thread carries out that operation on its own private data.

The number of warps for a thread block can be calculated as follows:

$$warps\ per\ block = ceil\left(\frac{threads\ per\ block}{warp\ size}\right)$$

thus, there's always a discrete number of warps per thread block, and a warp can't be split between blocks. If a block size is not an even multiple of warp size, some threads in the last warp are left inactive; while unused, they still consume SM resources (like registers).

GPUs lack the complex branch prediction hardware that CPUs have. Because all threads in a warp must execute identical instructions on the same cycle, this could become a problem if threads in the same warp take different paths: this condition is referred to as warp divergence, and can cause significantly degraded performance, because the warp serially executes each branch path, disabling threads that don't take that path.

Resource partitioning

The local execution context of a warp mainly consists of program counters, registers and shared memory.

The execution context of each warp processed by an SM is maintained on-chip during the entire lifetime of the warp, so that switching from one execution context to another has no cost.

Each SM has a set of 32-bit registers stored in a register file that are partitioned among threads, and a fixed amount of shared memory that is partitioned among blocks. The number of blocks and warps that can simultaneously reside on an SM for a given kernel depends on the number of

registers and amount of shared memory available on the SM and required by the kernel.

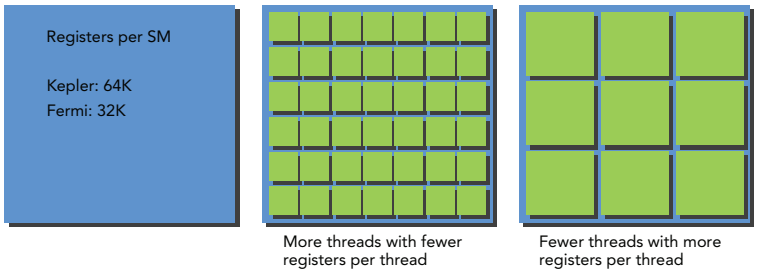


Figure 2.9: Warps on an SM with varying registers.[CGM14]

Fig. 2.9 illustrates that when a kernel consumes more registers, fewer warps can be placed on a SM; by reducing the *register pressure* more warps will be processed simultaneously.

Resource availability generally limits the number of resident thread blocks per SM. The number of registers and the amount of shared memory per SM vary for devices of different *compute capability*, which is a way to classify each GPU based on several characteristics that depend on the hardware architecture. For example, devices with compute capability 2.1 can have a maximum of 48 concurrent active warps per multiprocessor, while devices with capability 3.0 or greater increase this value to 64.

A thread block is called an *active block* when compute resources, such as registers and shared memory, have been allocated to it; the warps it contains are called *active warps*. Active warps can be further classified in: selected warp, stalled warp and eligible warp. The warp schedulers on an SM select active warps on every cycle and dispatch them to execution units. A warp that is actively executing is called a *selected warp*. If an active warp is ready for execution but not currently executing, it is an *eligible warp*. If a warp is not ready for execution, it is a *stalled warp*.

In order to maximise GPU utilisation, a large number of active warps is required.

Latency hiding

An shared multiprocessor relies on thread-level parallelism to maximise utilisation of its functional units: utilisation is therefore directly linked to the number of resident warps. The number of clock cycles between an instruction being issued and being completed is defined as instruction *latency*. Full compute resource utilisation is achieved when all warp schedulers have an eligible warp at every clock cycle: this ensures that the latency of each instruction can be hidden by issuing other instructions in other resident warps.

Latency hiding is very important in CUDA programming. CPU cores are designed to minimize latency for one or two threads at a time, whereas GPUs are designed to handle a large number of concurrent and lightweight threads in order to maximise throughput.

Latency hiding depends on the number of active warps per SM, which is implicitly determined by the execution configuration and resource constraints (registers and shared memory usage in a kernel). Choosing an optimal execution configuration is a matter of finding a balance between latency hiding and resource utilisation.

Occupancy

Instructions are executed sequentially within each thread: when one warp stalls, the SM switches to executing other eligible warps. It's ideal to have enough warps to keep the cores of the device occupied.

Occupancy is the ratio of active warps to maximum number of warps, per SM:

$$occupancy = \frac{active\ warps}{maximum\ warps}$$

Occupancy focuses exclusively on the number of concurrent threads or warps per SM. However, full occupancy is not the only goal for performance optimization, as there are also many other factors to examine for performance tuning: no single metric (occupancy, throughput, efficiency, etc.) can prescribe optimal performance.

3. Theoretical background

3.1 Homogeneous Isotropic Turbulence

Turbulent flows are of fundamental importance in engineering, but are also the most difficult king of flows to study.

In a turbulent flow, the velocity $\boldsymbol{U}(\boldsymbol{x}, t)$ is a time-dependent random vector, meaning that the random variable doesn't have the same value every time an experiment is repeated under the same conditions.

The Navier-Stokes equations are deterministic, yet they produce random solutions. This happens because there are perturbations on initial conditions, boundary conditions and flow properties, and turbulent flows show extreme sensitivity to these perturbations.

The random velocity field can be divided in two components (*Reynolds decomposition*):

$$\boldsymbol{U}(\boldsymbol{x}, t) \equiv \langle \boldsymbol{U}(\boldsymbol{x}, t) \rangle + \boldsymbol{u}(\boldsymbol{x}, t),$$

which are the mean velocity $\langle \boldsymbol{U} \rangle$ and the *fluctuating* velocity $\boldsymbol{u}$.

The random velocity field $\boldsymbol{U}(\boldsymbol{x}, t)$ is *statistically homogeneous* if all statistics are invariant for a shift in position (translations); if it's true, then the mean velocity $\langle \boldsymbol{U} \rangle$ is uniform and, with an appropriate choice of reference system, it can be set to zero.

If a random velocity field $\boldsymbol{U}(\boldsymbol{x}, t)$ is invariant under translations (statistically homogeneous), and also is statistically invariant under rotations and reflections of the coordinate system, then it's *statistically isotropic*.

Considering the complexity of turbulent flows, it's useful to exploit these statistical properties and consider a simple flow that is easier to study, in order to obtain a better insight in turbulent phenomena. *Homogeneous isotropic turbulence* is the simplest class of turbulent flows to study using direct numerical simulation (as it's done in this thesis), thanks to the previous properties; since there's no production of turbulence, the turbulence is decaying due to dissipation. A good approximation of this turbulent flow can be achieved in wind-tunnel experiments with grid turbulence.

3.2 The energy cascade

Lewis F. Richardson in 1922 [Ric22] proposed the concept of an *energy cascade*, in which turbulence is composed of *eddies* of size ℓ, each with a

characteristic velocity $u(\ell)$ and time scale $\tau(\ell)$. An "eddy" doesn't have a precise definition, but can be considered to be a turbulent motion localised within a particular region of space and that is moderately coherent within this region; a region occupied by a large eddy can also contain more smaller eddies.

Largest eddies are characterised by the length scale ℓ_0 which is comparable to the flow scale $\mathcal{L}$, and have velocities $u_0 \equiv u(\ell_o)$ with comparable order to the r.m.s (root-mean square) of turbulence intensity u_{rms} which is comparable to $\mathcal{U}$; they have a large Reynolds number (ratio between inertial and viscous forces), so the direct effects of viscosity are small.

Richardson's idea is that large eddies are unstable and break up, transferring their energy to smaller eddies, which then experience the same process. This transfer of energy towards smaller eddies continues until the Reynolds number is sufficiently small that the eddy motion is stable and viscous forces are effective in dissipating the kinetic energy.

Richardson's theory left many questions open for debate, and Kolmogrov in 1941 [Kol41] tried to answer some of them with a theory in the form of three hypotheses:

- *local isotropy*: at sufficiently high Reynolds number, the small-scale turbulent motions ($\ell \ll \ell_0$) are statistically isotropic;
- *first similarity*: at sufficiently high Reynolds number, the statistics of the small-scale motions ($\ell \ll \ell_{EI}$) have a universal form that is uniquely determined by ν and ε;
- *second similarity*: at sufficiently high Reynolds number, the statistics of the motions of scale ℓ in the range $\ell_0 \gg \ell\eta$ have a universal form that is uniquely determined by ε, independent of ν.

A consequence of the theory is that the length scale domain is divided in regions as illustrated in Fig. 3.1; the *energy-containing range* gets his name because it contains the bulk of energy, which is in the larger eddies in the size range $\ell_{EI} = \ell_0/6 < \ell < 6\ell_0$.

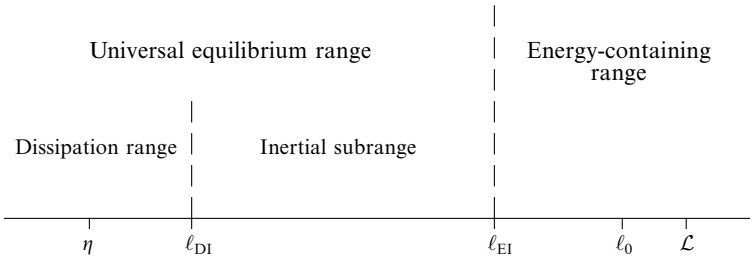


Figure 3.1: Lengthscales and ranges at high Reynolds number. [Pop00]

In general, large eddies are anisotropic and are affected by the boundary conditions of the flow; Kolmogrov's first hypothesis states that the directional biases of large scales are lost in the chaotic scale-reduction process. The term local isotropy means only at small scales. It's useful to introduce a lengthscale ℓ_{EI} (with $\ell_{EI} \approx \ell_0/6$) to divide the anisotropic large eddies ($\ell > \ell_{EI}$) from the isotropic small ones ($\ell < \ell_{EI}$).

Kolmogrov's second hypothesis states that also the informations about the geometry of large eddies are lost. In the energy cascade (for $\ell < \ell_{EI}$) the two dominant processes are the transfer of energy to smaller scales and viscous dissipation; the important parameters are the rate at which the small scales receive energy ($\mathcal{T}_{EI}$) and the kinematic viscosity ν. The dissipation rate is determined by the energy transfer rate, so they are nearly equal $\varepsilon \approx \mathcal{T}_{EI}$. The size range $\ell < \ell_{EI}$ is called *universal equilibrium range*; there, the time scales $\ell/u(\ell)$ are small compared to ℓ_0/u_0, thus small eddies can adapt quickly to maintain a dynamic equilibrium with the energy-transfer rate $\mathcal{T}_{EI}$ imposed by the large eddies. Give the parameters ε and ν, there are unique length, velocity and time scales that can be formed called *Kolmogrov scales*:

$$\eta \equiv (\nu^3/\varepsilon)^{1/4}$$
$$u_\eta \equiv (\nu\varepsilon)^{1/4}$$
$$\tau_\eta \equiv (\nu/\varepsilon)^{1/2}$$

The Reynolds number obtained from these scales is unity ($\eta u_\eta/\nu = 1$), indicating that the cascade goes to such small eddies that the dissipation is effective.

The ratio of the smallest to largest scales are determined from the definition of Kolmogrov scales and from the scaling $\varepsilon \sim u_0^3/\ell_0$:

$$\eta/\ell_0 \sim Re^{-3/4}$$
$$u_\eta/u_0 \sim Re^{-1/4}$$
$$\tau_\eta/\tau_0 \sim Re^{-1/2}$$

This confirms that at high Reynolds number, the velocity and time scales of small eddies are small compared to those of the large eddies (u_0 and τ_0).

The ratio η/ℓ_0 decreases with increasing Re, as a consequence at sufficiently high Re there is a range of scales ℓ that are very small compared to ℓ_0 but very large compared to η; eddies in this range are much bigger than the dissipative eddies so their Reynolds number is large and so their motions is little affected by viscosity: this is the third hypothesis. It's convenient to introduce a length scale ℓ_{DI} (with $\ell_{DI} = 60\eta$) to split the universal equilibrium range ($\ell < \ell_{EI}$) in two subranges: the *inertial subrange* ($\ell_{EI} > \ell > \ell_{DI}$) and the *dissipation range*. Only the motions in the dissipation range experience significant viscous effects and are responsible for almost all the dissipation.

Given an eddy size ℓ in the inertial subrange, characteristic velocity and time scales for the eddy are formed from ε and ℓ:

$$u(\ell) = (\ell\varepsilon)^{1/3} = u_\eta(\ell/\eta)^{1/3} \sim u_0(\ell/\ell_0)^{1/3}$$
$$\tau(\ell) = (\ell^2/\varepsilon)^{1/3} = \tau_\eta(\ell/\eta)^{2/3} \sim \tau_0(\ell/\ell_0)^{2/3}$$

As a consequence of the third hypothesis, in the inertial subrange the velocity and time scales decreases as ℓ increases.

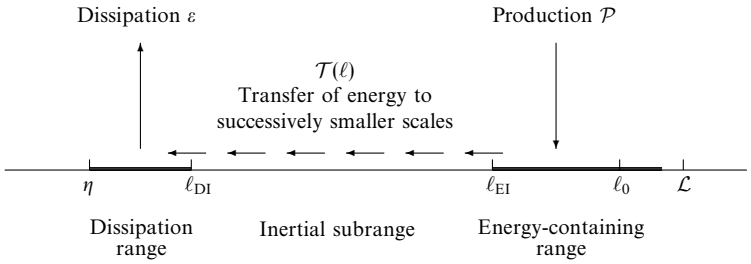


Figure 3.2: Energy cascade at high Reynolds number. [Pop00]

If the energy transfer process is realised mainly by eddies of size similar to ℓ, then $\mathcal{T}(\ell)$ is of order $u(\ell)^2/\tau(\ell)$; this ratio, using the previous identities, corresponds to ε. The important conclusions is that, for ℓ in the inertial range, $\mathcal{T}(\ell)$ is independent of ℓ, and is equal to ε. Hence it's possible to write, for $\ell_{EI} > \ell > \ell_{DI}$:

$$\mathcal{T}_{EI} = \mathcal{T}(\ell) = \mathcal{T}_{DI} = \varepsilon$$

The rate of energy transfer from the large scales $\mathcal{T}_{EI}$ determines the constant rate of energy transfer through the inertial subrange $\mathcal{T}(\ell)$,where dissipation can be ignored. This is illustrated in Fig. 3.2.

3.2.1 The energy spectrum

To determine the distribution of turbulent kinetic energy over the eddies of different size, for homogeneous turbulence it's possible to consider the energy spectrum function $E(\kappa)$.

The *two-point correlation* (two-point, one time auto-covariance) is the simplest statistic containing informations regarding the spatial structure of a random field:

$$R_{ij}(\boldsymbol{r}, \boldsymbol{x}, t) \equiv \langle u_i(\boldsymbol{x}, t) u_j(\boldsymbol{x} + \boldsymbol{r}, t) \rangle \tag{3.1}$$

The *velocity spectrum tensor* is the Fourier transform of the two-point correlation:

$$\Phi_{ij}(\boldsymbol{\kappa}, t) = \frac{1}{(2\pi)^3} \iiint_{-\infty}^{+\infty} e^{-i\boldsymbol{\kappa}\cdot\boldsymbol{r}} R_{ij}(\boldsymbol{r}, t) d\boldsymbol{r}.$$

The *energy spectrum function* is defined as:

$$E(\kappa,t) \equiv \iiint_{-\infty}^{+\infty} \frac{1}{2}\Phi_{ii}(\boldsymbol{\kappa},t)\delta(|\boldsymbol{\kappa}|-\kappa)d\boldsymbol{\kappa},$$

which can be imagined as $\Phi_{ij}(\boldsymbol{\kappa},t)$ without all directional informations. Integrated over all wavenumbers κ it yields:

$$\int_0^\infty E(\kappa,t)d\kappa = \frac{1}{2}R_{ii}(0,t) = \frac{1}{2}\langle u_i u_i\rangle.$$

Therefore $E(\kappa,t)d\kappa$ represents the contribution to the turbulent kinetic energy from all the modes with $|\boldsymbol{\kappa}|$ in the range $\kappa \leq |\boldsymbol{\kappa}| \leq \kappa + d\kappa$.

Motions of lengthscale ℓ correspond to wavenumber $\kappa = 2\pi/\ell$; the energy in the wavenumber range (κ_a, κ_b) can be expressed as:

$$k_{(\kappa_a,\kappa_b)} = \int_{\kappa_a}^{\kappa_b} E(\kappa)d\kappa. \tag{3.2}$$

The contribution to the dissipation rate ε from the motions in the range (κ_a, κ_b) is:

$$\varepsilon_{(\kappa_a,\kappa_b)} = \int_{\kappa_a}^{\kappa_b} 2\nu\kappa^2 E(\kappa)d\kappa. \tag{3.3}$$

Exploiting Kolmogrov's second hypothesis, in the universal equilibrium range ($\kappa > \kappa_{EI}$) the energy spectrum is an universal function of ε and ν. From the third hypothesis it follows that, in the inertial range ($\kappa_{EI} < \kappa < \kappa_{DI}$), the energy spectrum is defined by the equation:

$$E(\kappa) = C\varepsilon^{2/3}\kappa^{-5/3},$$

where C is a universal constant ($C = 1.5$). This is Kolmogrov's -5/3 law, and is very well verified by experimental data. Using this law it's possible to demonstrate that the amount of energy in the high wavenumbers decreases as $k_{(\kappa,\infty)} \sim \kappa^{-2/3}$ as κ increases, whereas the dissipation in the low wavenumbers decreases as $\varepsilon_{(0,\kappa)} \sim \kappa^{4/3}$ as κ decreases towards zero.

Although this law applies only to the inertial range, the previous observations are consistent with the fact that the bulk of energy is in the large scales ($\ell > \ell_{EI}$ or $\kappa < 2\pi/\ell_{EI}$), and that the bulk of dissipation is in the small scales ($\ell < \ell_{DI}$ or $\kappa > 2\pi/\ell_{DI}$).

3.3 The Taylor scales

The Kolmogrov hypotheses have no direct connection to the Navier-Stokes equations. Despite this fact, it's possible to extract some informations about

the energy cascade from the Navier-Stokes equations by following the first attempts made by Taylor in 1935 [Tay35] and Howarth in 1938 [vK38], which are based on the two-point correlation.

The flow is homogeneous isotropic turbulence with zero mean velocity, r.m.s. velocity $u_{rms}(t)$ and dissipation rate $\varepsilon(t)$; thanks to homogeneity the two-point correlation (Eq. (3.1)) does not depend from $\boldsymbol{x}$, and at the origin of the reference system its value is:

$$R_{ij}(0,t) = \langle u_i u_j \rangle = u_{rms}^2 \delta_{ij}$$

The isotropy property allows the two-point correlation to be expressed as a function of two scalar functions $f(r,t)$ and $g(r,t)$:

$$R_{ij}(\boldsymbol{r},t) = u_{rms}^2 \left(g(r,t)\delta_{ij} + [f(r,t) - g(r,t)] \frac{r_i r_j}{r^2} \right). \tag{3.4}$$

For a cartesian reference system chosen in a way that $\boldsymbol{r}$ is in the x_1 direction ($\boldsymbol{r} = \boldsymbol{e}_1 r$), the previous equation becomes:

$$\begin{aligned}
&R_{11}/u_{rms}^2 = f(r,t) = \langle u_1(\boldsymbol{x} + \boldsymbol{e}_1 r, t) u_1(\boldsymbol{x}, t) \rangle / \langle u_1^2 \rangle \\
&R_{22}/u_{rms}^2 = g(r,t) = \langle u_2(\boldsymbol{x} + \boldsymbol{e}_1 r, t) u_2(\boldsymbol{x}, t) \rangle / \langle u_2^2 \rangle \\
&R_{33} = R_{22} \\
&R_{ij} = 0 \qquad \text{for} \quad i \neq j
\end{aligned}$$

The functions $f(r,t)$ and $g(r,t)$ are, respectively, the longitudinal and transverse autocorrelation functions. The continuity equation implies that $\partial R_{ij}/\partial r_j$, which applied to Eq. (3.4) results in:

$$g(r,t) = f(r,t) + \frac{1}{2} \frac{\partial}{\partial r} f(r,t).$$

The conclusion is that, in homogeneous isotropic turbulence, the two-point correlation $R_{ij}(\boldsymbol{r},t)$ is function of the longitudinal autocorrelation function $f(r,t)$ alone. Measurements of $f(r,t)$ in nearly isotropic grid-generated turbulence are illustrated in Fig. 3.3.

From $f(r,t)$ and $g(r,t)$ it's possible to derive two longitudinal and two transverse lengthscales.

The *longitudinal integral scale* is obtained from $f(r,t)$:

$$L_{11}(t) \equiv \int_0^\infty f(r,t) dr,$$

which is the area under the curve of $f(r,t)$; observing Fig. 3.3, it's clear that this scale increases with time in grid turbulence. L_{11} is characteristic of the larger eddies.

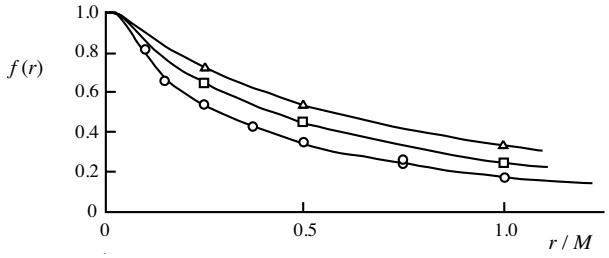


Figure 3.3: Measurements of the longitudinal autocorrelation function in grid turbulence; higher curves have higher distance from the grid; M is the mesh spacing of the grid. (Comte-Bellot and Corrsin (1971))

In isotropic turbulence, the *transverse integral scale* is:

$$L_{22}(t) \equiv \int_0^\infty g(r,t)dr,$$

and it's exactly half of $L_{11}(t)$.

From $f(r,t)$ it's also possible to obtain the *longitudinal Taylor microscale*:

$$\lambda_f(t) = \left[-\frac{1}{2}f''(0,t)\right]^{-1/2},$$

which is a real function because $f''(0,t) = (\partial^2 f/\partial r^2)_{r=0} \leq 0$. Geometrically $\lambda_f(t)$ represents the position where the parabola osculating $f(r,t)$ at $r=0$ intersects the r axis.

The *transverse Taylor microscale* is defined as:

$$\lambda_g(t) = \left[-\frac{1}{2}g''(0,t)\right]^{-1/2},$$

and in isotropic turbulence it's equal to $\lambda_f(t)/\sqrt{2}$. From these two equations and the relation $\varepsilon = 15\nu\langle(\partial u_1/\partial x_1)^2\rangle$, it follow that the dissipation is:

$$\varepsilon = \frac{15\nu u_{rms}^2}{\lambda_g^2}. \tag{3.5}$$

Taylor microscales are related to Kolmogrov's scales. Defined the length scales characterising large eddies as $L \equiv k^{3/2}/\varepsilon$, the corresponding Reynolds number is:

$$Re_L \equiv \frac{k^{1/2}L}{\nu} = \frac{k^2}{\varepsilon\nu}.$$

Then the microscales are:

$$\lambda_g/L = \sqrt{10}Re_L^{-1/2} \tag{3.6}$$

$$\eta/L = Re_L^{-3/4} \tag{3.7}$$

$$\lambda_g = \sqrt{10}\eta^{2/3}L^{1/3} \tag{3.8}$$

From this relations it's clear that, at high Reynolds number, λ_g has a size intermediate between the Kolmogrov's (η) and large (L) scales.

Despite the lack of clear physical interpretation, the Taylor scale is particularly used in grid turbulence, especially with the Taylor-scale Reynolds number:

$$R_\lambda \equiv \frac{u_{rms}\lambda_g}{\nu}.$$

From Eq. (3.6) it's possible to obtain the following relation:

$$R_\lambda = \left(\frac{20}{3}Re_L\right)^{1/2}.$$

Finally, from Eq. (3.5) it can be observed that the ratio:

$$\lambda_g/u_{rms} = (15\nu/\varepsilon)^{1/2} = \sqrt{15}\tau_\eta$$

correctly characterise the time scale of small eddies.

3.3.1 The Kármán-Howarth equation

In 1938, von Kármán and Howarth [vK38] obtained an evolution equation for the longitudinal autocorrelation function $f(r,t)$. Starting from the two-point correlation tensor and deriving it in time, using the Navier-Stokes equations and the isotropy property, it's possible to obtain the *Kármán-Howarth equation*:

$$\frac{\partial}{\partial t}(u_{rms}^2 f(r,t)) - \frac{u_{rms}^3}{r^4}(r^4\bar{k}) = \frac{2\nu u_{rms}^2}{r^4}\frac{\partial}{\partial r}\left(r^4\frac{\partial f(r,t)}{\partial r}\right),$$

where $\bar{k}$ is a two-point triple velocity correlation (third order moment):

$$\bar{k}(r,t) = \bar{S}_{111}(\boldsymbol{e}_1 r,t)/u_{rms}^3 = \langle u_1(\boldsymbol{x},t)u_1(\boldsymbol{x},t)u_1(\boldsymbol{x}+\boldsymbol{e}_1,t)\rangle/u_{rms}^3,$$

representing inertial processes, which effectively realise the energy transfer from larger to smaller scale in Richardson's theory.

The Kármán-Howarth equation involves two unknown functions: $f(r,t)$ and $\bar{k}(r,t)$. This fact highlights the closure problem of turbulence: it could be possible to write and evolution equation for $\bar{k}(r,t)$, but it would contain fourth order moments which are unknown and only obtainable through an evolution equation for them, and so on.

If the velocity field is well described by a Gaussian distribution, then $\bar{k}(r,t)$ would be zero, like it's typical for all third moments. This means that the energy cascade occurs only if the velocity field is non-Gaussian.

Since isotropic has no forcing, for long periods it's bound to decay, the Reynolds number decreases and eventually inertial effects becomes negligible. In the *final period of decay*, Batchelor and Townsend proved that

the Kármán-Howarth equation, without inertial terms, admits a self-similar solution:

$$f(r,t) = e^{-r^2/(8\nu t)},$$

which is also valid to very low Reynolds number, and is very well verified by experimental data [BT48].

While the Kármán-Howarth equation gives a model for $f(r,t)$, it doesn't provide a clear description of the dynamics of the scales and of the processes involved in the energy cascade. To obtain better insights it's useful to examine the Navier-Stokes equations in wavenumber domain.

4. Numerical simulations of turbulent flows

4.1 The problem

The study of turbulent flows is notoriously difficult because of the lack of a simple analytic theory, although efforts are continuously made in order to solve this issue. To this day it's still an unsolved problem, so the hope is to use the ever-increasing power of computers to calculate the properties of turbulent flows.

There are several reasons that contribute to the inability to develop an accurate theory or model. In the first place, the velocity field $\boldsymbol{U}(\boldsymbol{x},t)$ is 3D, time-dependent and random. The turbulent motions develop over a large range of time scales and length scales; the largest turbulent motions have the dimensions of the characteristic width of the flow, therefore are affected by the boundary geometry and hence are not universal.

Relative to the largest scales, the Kolmogrov time scale decreases as $\mathrm{Re}^{-1/2}$ and the Kolmogrov lengthscales as $\mathrm{Re}^{-3/4}$. In wall-bounded flows, the most energetic motions, responsible for the peak turbulence production, scale with the viscous length δ_v, which is small compared to the outer scale δ, and which decreases (relative to δ) approximately as $\mathrm{Re}^{-4/5}$.

Another difficulty comes from the non-linear convective tern in the Navier-Stokes equations and, in greater part, from the pressure-gradient term; when it's expressed in terms of velocity via the solution to the Poisson equation, the pressure-gradient term is both non-linear and non-local.

4.2 Different approaches

Two main categories are turbulent-flow *simulations* and *turbulence models*. In a turbulent-flow simulation the equations are solved for a time-dependent velocity field that partially represent the velocity field $\boldsymbol{U}(\boldsymbol{x},t)$ for one realisation of the turbulent flow. In a turbulence model the equations are solved for some mean quantities, for example $\langle \boldsymbol{U} \rangle$, $\langle u_i u_j \rangle$ and ε.

Two simulation approaches are direct numerical simulation (DNS, the object of this study) and large-eddy simulation (LES). In DNS, the Navier-Stokes equations are solved to determine $\boldsymbol{U}(\boldsymbol{x},t)$ for one realisation of the flow. Because all length scales and and time scales have to be resolved, this approach is computationally expensive; since the computational cost increases as Re^3, DNS are restricted to flows with low-to-moderate Reynolds

number. In LES, equations are solved for a 'filtered' velocity field $\overline{\boldsymbol{U}}(\boldsymbol{x},t)$, which represents turbulent motions of larger scale; the equations solved include a model for the influence of the smaller-scale motions which are not directly represented.

There are other approaches called RANS (Reynolds-averaged Navier-Stokes), since they involve the solution of the Reynolds equation to determine the mean velocity field $\langle \boldsymbol{U} \rangle$. In one of these approaches, the Reynolds stresses are obtained from a turbulent-viscosity model; the turbulent viscosity can be obtained from an algebraic relation (like in the mixing-length model) or it can be obtained from turbulent quantities such as k and ε for which modelled transport equations are solved. In Reynolds-stress models, modelled transport equations are solved for the Reynolds stresses, thus avoiding the need for a turbulent viscosity. The mean velocity $\langle \boldsymbol{U} \rangle$ and the Reynolds stresses $\langle u_i u_j \rangle$ are the first and second moments of the Eulerian PDF (probability density function) of velocity $f(\boldsymbol{V};\boldsymbol{x},t)$. In PDF methods, a model transport equations is solved for a PDF such as $f(\boldsymbol{V};\boldsymbol{x},t)$.

Decide between approaches

Historically, many approaches have been proposed and many are currently in use. There is a broad range of turbulent flows, so it's appropriate to have a broad range of models that vary in complexity, accuracy and other attributes.

Here are the principal criteria that can be used to evaluate different approaches, with a focus on DNS:

- *level of description*: in DNS the flow is described by the instantaneous velocity $\boldsymbol{U}(\boldsymbol{x},t)$, from which all other information can be obtained (turbulent structures, multi-time and multi-point statistics, etc.). A higher level of description can provide a more complete characterisation of turbulence, leading to more accurate models with wider applicability;

- *completeness*: complete means that the equations are free from flow dependent specifications. One flow is distinguished from another only by the specifications of material properties (ρ and ν) and of initial and boundary conditions. DNS is complete;

- *range of applicability*: not all approaches are applicable to all flows. For DNS, the computational requirements rise so steeply with Reynolds number that this approach is applicable only to flows of low to moderate Reynolds number;

- *accuracy*: in application to a particular flow, the accuracy of an approach can be determined by comparing calculations and experimental measurements. The numerical solution inevitably contains numerical

errors from a number of sources, but it's often dominated by spatial truncation error: the available computer resources don't allow a sufficiently fine grid spacing;

- *cost and ease of use*: the difficulty of performing a calculation depends both on the flow and on the model. The computational difficulty increases with the statistical dimensionality of the flow: it decreases if the flow is statistically stationary, and decreases further if boundary-layer equations can be used. In DNS, the computational cost is a rapidly increasing function of the Reynolds number of the flow. The evolution of supercomputers is depicted in Fig. 4.1, predicting an improvement of 1000 times in a decade;

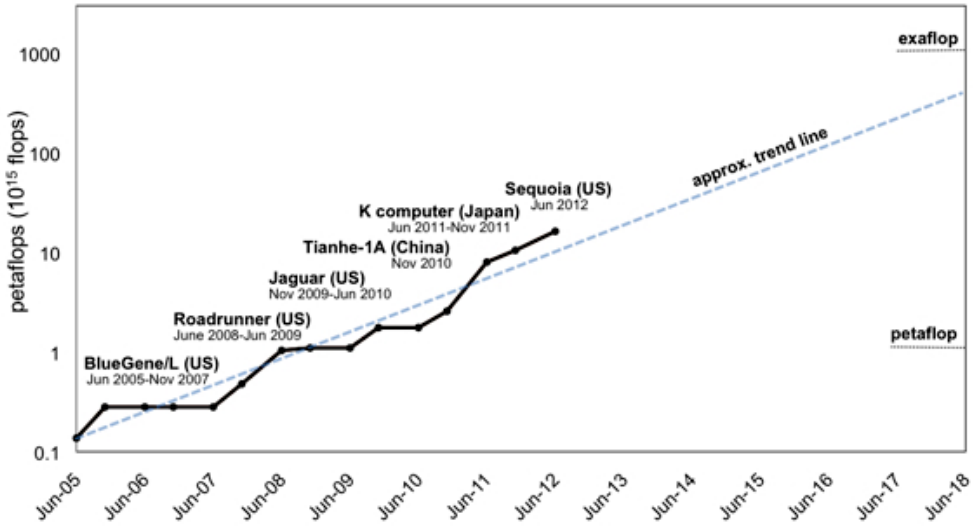


Figure 4.1: Speed of the fastest supercomputers (in petaflops, million of gigaflops) against the year of their introductions, with an estimated trend line predicting one exaflop by 2018 (adapted from [Dor12]).

4.3 Direct numerical simulations

Direct numerical simulation (DNS) consists in solving the Navier-Stokes equations, resolving all scales of motion, with initial and boundary conditions. The DNS approach was impossible to execute until the 1970s, when computers of sufficient power became available.

Conceptually it's the simplest approach and, in the situations where it can be applied, it's unrivalled in accuracy and level of description obtained. Due to extremely high computational cost with increasing Reynolds number, its applicability is limited to flows with low to moderate Reynolds numbers.

For homogeneous turbulence, *pseudo-spectral* methods (pioneered by Orszag & Patterson [OP72] and Rogallo [Rog81]) are the preferred numerical approach to the solution, thanks to their superior efficiency.

In a DNS of homogeneous isotropic turbulence, the solution domain is a cube of side ℓ, and the velocity field $\hat{u}(\boldsymbol{\kappa}, t)$ is represented as a finite Fourier series. In total, N^3 wavenumbers are represented; N represents the size of the simulation, and consequently the Reynolds number that can be achieved. The lowest non-zero wavenumber is $\kappa_0 = 2\pi/\ell$, and the N^3 represented are:

$$\boldsymbol{\kappa} = \kappa_0 \boldsymbol{n} = \kappa_0 (l\boldsymbol{e}_1 + m\boldsymbol{e}_2 + n\boldsymbol{e}_3),$$

for integer values of *l,m,n* between $-\frac{N}{2} + 1$ and $\frac{N}{2}$. In each direction, the largest wavenumber is:

$$\kappa_{max} = \frac{1}{2} N \kappa_0 = \frac{\pi N}{\ell}. \tag{4.1}$$

The spectral representation coincides with representing $\boldsymbol{u}(\boldsymbol{x}, t)$ in physical space on an N^3 grid with uniform spacing:

$$\Delta x = \frac{\ell}{N} = \frac{\pi}{\kappa_{max}}.$$

A *spectral* method consists in advancing the Fourier modes $\hat{u}(\boldsymbol{\kappa}, t)$ in time steps Δt according to the Navier–Stokes equations represented in wavenumber space (Eq. (4.11)). The triad requires about N^6 operations; to avoid this high cost, in *pseudo-spectral* methods the nonlinear terms in the Navier-Stokes equations are evaluated differently: the velocity field is transformed into physical space, they are multiplied together and then transformed back in wavenumber domain. This procedure employs only $N^3 logN$ operations, thus improving greatly the computational efficiency compared to spectral methods. However, an *aliasing error* is introduced and needs to be removed.

4.3.1 Computational cost

The computational cost is mostly determined by the resolution requirements. The domain size ℓ must be large enough to represent the energy-containing motions, and the grid spacing Δx must be small enough to resolve the dissipative scales. In addition, the time step Δt used to advance the solution is limited by considerations of numerical accuracy.

For isotropic turbulence, a reasonable lower limit on ℓ is eight integral length scales ($\ell = 8L_{11}$); it means that the lowest wavenumber is:

$$\kappa_0 L_{11} = \frac{2\pi}{\ell} \frac{\ell}{8} = \frac{\pi}{4} \approx 0.8.$$

The Fourier representation of the velocity field requires periodic boundary conditions on the solution domain. In Fig. 4.2 it's evident the effect

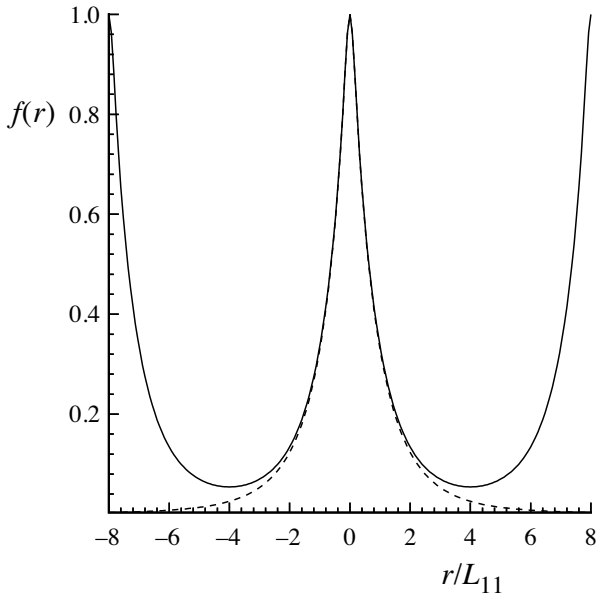


Figure 4.2: Effect of periodicity on the longitudinal autocorrelation function $f(r)$. Dashed line: $f(r)$ for the model spectrum; solid line: $f(r)$ for the periodic velocity field with $\ell = 8L_{11}$. [Pop00]

of periodic boundary conditions on the longitudinal autocorrelation function $f(r)$. With infinite low-wavenumber resolution of the model spectrum ($\ell/L_{11} \to \infty$), the autocorrelation function tends to zero at the boundaries of the domain; for $\ell/L_{11} = 8$, just after $|r|/L_{11}$ the differences are already obvious.

The resolution of the smallest, dissipative motions (characterised by the Kolmogrov scale η) requires a sufficiently small grid spacing $\Delta x/\eta$ (which corresponds to a sufficiently large maximum wavenumber $\kappa_{max}\eta$. The dissipation spectrum is very small for values of $\kappa\eta$ greater than 1.5, so it's a good criterion for achieving adequate resolution of the smallest scales. The corresponding grid spacing is:

$$\frac{\Delta x}{\eta} = \frac{\pi/\kappa_{max}}{1.5\ \kappa_{max}} = \frac{\pi}{1.5} \approx 2.1.$$

The two spatial resolution requirements found determine the necessary number of Fourier modes (or grid nodes in physical space) N as a function of the Reynolds number. Manipulating Eq. (4.1) yields:

$$N = 2\frac{\kappa_{max}}{\kappa_0} = 2\frac{\kappa_{max}\eta}{\kappa_0 L_{11}}\left(\frac{L_{11}}{L}\right)\left(\frac{L}{\eta}\right) = \frac{12}{\pi}\left(\frac{L_{11}}{L}\right)\left(\frac{L}{\eta}\right), \qquad (4.2)$$

where $L \equiv k^{3/2}/\varepsilon$.

Using the model spectrum to obtain L_{11}/L, the value of N obtained is shown as a function of the Reynolds number in Fig. 4.3. At high Reynolds number, L_{11}/L has the asymptotic value of *0.43*, thus the previous equation becomes:

$$N \sim 1.6 \left(\frac{L}{\eta}\right) = 1.6 Re_L^{3/4} \approx 0.4 R_\lambda^{3/2}, \tag{4.3}$$

while the total number of modes increases as

$$N^3 \sim 4.4 Re_L^{9/4} \approx 0.06 R_\lambda^{9/2}$$

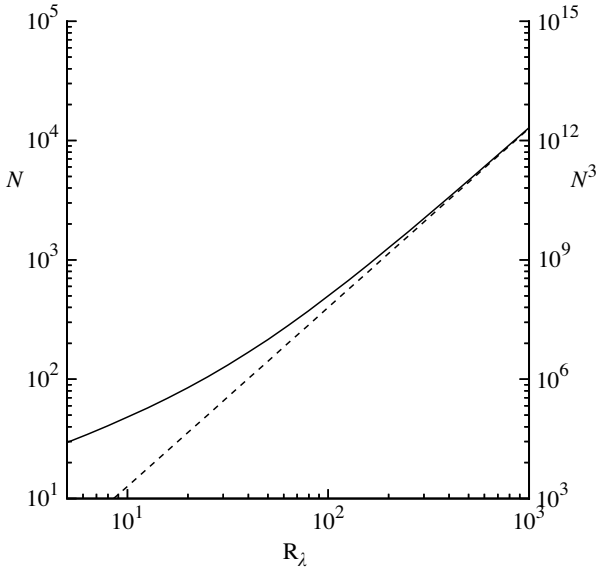


Figure 4.3: Number of Fourier modes (or grid nodes) N in each direction required for a proper resolution of isotropic turbulence. Dashed line: asymptote, Eq. (4.3); solid line: Eq. (4.2). [Pop00]

For the temporal advancement of the solution, in order to be accurate it's necessary that a fluid particle moves only a fraction of the grid spacing Δx in a time step Δt. The *Courant number* imposed is approximately:

$$C = \frac{k^{1/2}\Delta t}{\Delta x} = \frac{1}{20}.$$

The time step Δt determined by the Courant number is a restriction imposed by the numerical methods employed. The intrinsic restriction on Δt imposed by the turbulence is that $\Delta t/\tau_\eta$ should be small.

The duration of a simulation usually is in the order of four times the turbulence time scale ($\tau = k/\varepsilon$), so the number of time steps required is:

$$M = \frac{4\tau}{\Delta t} = \frac{4k/\varepsilon}{\Delta x/(20k^{1/2})} = \frac{80k^{3/2}}{\varepsilon \Delta x} = \frac{80L}{\Delta x} = \frac{80L}{\pi\eta/1.5} \approx 9.2 R_\lambda^{3/2}$$

Approximately, the number of floating point operations required for a simulation depends on the number of Fourier modes (or grid nodes) and on the number of steps, that is it depends on N^3M (mode-steps). The preceding results yields:

$$N^3M \sim 160Re_L^3 \approx 0.55R_\lambda^6,$$

which exhibit a very steep rise with the Reynolds number.

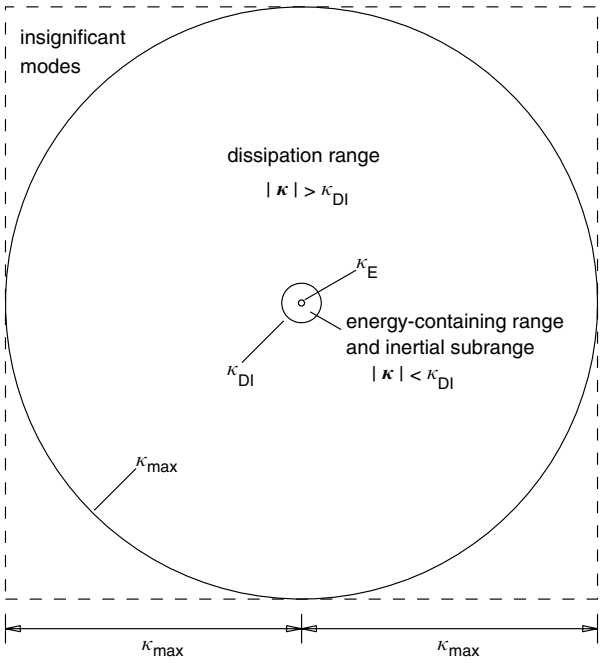


Figure 4.4: Solution domain in wavenumber space for a pseudo-spectral DNS of isotropic turbulence, at $R_\lambda = 70$. [Pop00]

It's interesting to examine the distribution of computational effort among the scales of turbulent motion. In the three-dimensional wavenumber domain, the represented Fourier modes are virtually contained in a cube of side $2\kappa_{max}$. In a well resolved simulation (for example with $\kappa_{max}\eta = 1.5$), only the modes inside the sphere of radius κ_{max} are dynamically significant. As evident in Fig. 4.4, the dissipative range corresponds to the spherical shell of wavenumber $|\boldsymbol{\kappa}|$ between κ_{DI} and κ_{max}, where $\kappa_{DI} = 0.1/\eta$ is the wavenumber of the largest dissipative motions. The sphere of radius κ_{DI} contains the energy-containing motions and, at sufficiently high Reynolds number, the inertial-range motions. In Fig. 4.4, the inner sphere of radius κ_E corresponds to the peak in energy spectrum for isotropic turbulence at $R_\lambda = 70$: when the Reynolds number increases, its radius becomes smaller.

It's possible to prove that 99.98% of the modes represented have wavenumbers $|\boldsymbol{\kappa}| > \kappa_{DI}$, and less than 0.02% of the modes represent in the energy-containing range or in the inertial subrange.

It's clear that, where it can be applied, DNS provides a level of description and accuracy that can't be matched by other numerical approaches, and also gives the opportunity to obtain certain statistics that are essentially impossible to obtain through experiments. However, not all DNS simulations corresponds to a realisable turbulent flow due to artificial modifications, incomplete resolution and non-physical boundary conditions.

What is even more evident is the drawback of DNS: massive computational cost, and the fact that it increases rapidly with the Reynolds number (approximately as Re^3). Since computer times are typically about 200 hours on a supercomputer, only flows with low or moderate Reynolds numbers can be simulated.

There is a contradiction between DNS and the objective of determining the mean velocity and energy containing motions in a turbulent flow. In DNS, over 99% of the effort goes toward the dissipation range (Fig. 4.4), and this effort increases strongly with the Reynolds number. By contrast, the mean flow and the statistics of the energy-containing motions exhibit only weak Reynolds-number dependences.

4.4 The pseudo-spectral method

4.4.1 Fourier series

A phenomenon like turbulence can be described in two domains. There's the physical domain with a characteristic quantity h function of the position $\boldsymbol{x}=\{x_1,x_2,x_3\}$, and the wavenumber domain with an amplitude $\hat{h}$ function of the wavenumber vector $\boldsymbol{\kappa}=\{\kappa_1,\kappa_2,\kappa_3\}$.

The Fourier series equation in the physical domain is:

$$h(\boldsymbol{x}) = \sum_{\kappa} \hat{h}(\boldsymbol{\kappa}) e^{i\boldsymbol{\kappa}\cdot\boldsymbol{x}}$$

where $\hat{h}(\boldsymbol{\kappa})$ are the Fourier series' coefficients, modal amplitudes (usually complex numbers); $e^{i\boldsymbol{\kappa}\cdot\boldsymbol{x}}$ are Fourier's modes (polynomials).

If control volume $\mathcal{V}$ is a cube of side wavenumber ℓ, discretised in each of the three directions with N equally spaced points, it's then possible to define the discrete Fourier transform (DFT) operator:

$$\hat{h}(\boldsymbol{\kappa}) = \mathcal{F}_{\boldsymbol{\kappa}}\{h(\boldsymbol{x})\} = \frac{1}{N^3} \sum_{\boldsymbol{x}} h(\boldsymbol{x}) e^{-i\boldsymbol{\kappa}\cdot\boldsymbol{x}} \tag{4.4}$$

that allows to pass from the physical space to the wavenumber space.

If the control volume $\mathcal{V}$ has finite dimensions, the application of Fourier series impose an artificial spatial periodicity to the function $h(\boldsymbol{x})$, with period equal to ℓ along the three directions, therefore:

$$h(\boldsymbol{x}) = h(\boldsymbol{x} + \boldsymbol{N}\ell)$$

for each vector of integer numbers $\boldsymbol{N}$.

By defining the vector $\boldsymbol{n}$=$\{l\boldsymbol{e}_1,m\boldsymbol{e}_2,n\boldsymbol{e}_3\}$ with $\{\boldsymbol{e}_1,\boldsymbol{e}_2,\boldsymbol{e}_3\}$ unit vectors of the cartesian orthogonal reference system, and the wavenumber with the lowest intensity κ_0=$\frac{2\pi}{\ell}$, the vector $\boldsymbol{\kappa}$ can be written as:

$$\boldsymbol{\kappa} = \kappa_0\boldsymbol{n} = \kappa_0 l\boldsymbol{e}_1 + \kappa_0 m\boldsymbol{e}_2 + \kappa_0 n\boldsymbol{e}_3 \tag{4.5}$$

from which:

$$e^{i\boldsymbol{\kappa}\cdot\boldsymbol{x}} = e^{i\kappa_0 l x_1} e^{i\kappa_0 m x_2} e^{i\kappa_0 n x_3}$$

The Fourier series benefits from several properties, of which:

- *linearity*: with α and β arbitrary constants:

$$\mathcal{F}_\kappa\{\alpha f(\boldsymbol{x}) + \beta g(\boldsymbol{x})\} = \alpha\mathcal{F}_\kappa\{f(\boldsymbol{x}) + \beta\mathcal{F}_\kappa\{g(\boldsymbol{x}) \tag{4.6}$$

- *conjugated symmetry*: if function $h(\boldsymbol{x})$ is real, $\hat{h}(\boldsymbol{\kappa})$ is Hermitian:

$$\hat{h}(\boldsymbol{\kappa}) = \hat{h}^*(-\boldsymbol{\kappa}) \tag{4.7}$$

  with $\hat{h}^*(-\boldsymbol{\kappa})$ conjugate complex of $\hat{h}(\boldsymbol{\kappa})$;

- *derivation*: the derivative operator gain a much simpler form in the Fourier dominion:

$$\mathcal{F}_\kappa\left\{\frac{\partial h(\boldsymbol{x})}{\partial x_j}\right\} = i\kappa_j\hat{h}(\boldsymbol{\kappa}). \tag{4.8}$$

  Proof: derivation of the Fourier series equation with respect to $\boldsymbol{x}$ gives:

$$\frac{\partial h(\boldsymbol{x})}{\partial x_j} = \sum_{\boldsymbol{\kappa}} i\kappa_j\hat{h}(\boldsymbol{\kappa})e^{i\boldsymbol{\kappa}\cdot\boldsymbol{x}} = \mathcal{F}_\kappa^{-1}\{i\kappa_j\hat{h}(\boldsymbol{\kappa})\},$$

  transforming with Fourier on both sides gives the final result:

$$\mathcal{F}_\kappa\left\{\frac{\partial h(\boldsymbol{x})}{\partial x_j}\right\} = \mathcal{F}_\kappa\left\{\mathcal{F}_\kappa^{-1}\{i\kappa_j\hat{h}(\boldsymbol{\kappa})\}\right\} = i\kappa_j\hat{h}(\boldsymbol{\kappa}).$$

It's possible to define the following operators:

- *convolution* of two functions:

$$f(\boldsymbol{x}) * g(\boldsymbol{x}) = \sum_{\boldsymbol{x}'} f(\boldsymbol{x}')g(\boldsymbol{x}-\boldsymbol{x}');$$

for the convolution theorem:

$$\begin{aligned} \mathcal{F}_\kappa\{f(\boldsymbol{x}) * g(\boldsymbol{x})\} &= \hat{f}(\boldsymbol{\kappa})\hat{g}(\boldsymbol{\kappa}) \\ \mathcal{F}_\kappa^{-1}\{f(\boldsymbol{\kappa}) * g(\boldsymbol{\kappa})\} &= f(\boldsymbol{x}) * g(\boldsymbol{x}) \end{aligned} \tag{4.9}$$

- *projection tensor* of a generic vector $\hat{\boldsymbol{G}}$ that can be decomposed in perpendicular and parallel component to the wavenumber vector $\boldsymbol{\kappa}$ as $\hat{\boldsymbol{G}} = \hat{\boldsymbol{G}}^{\parallel} + \hat{\boldsymbol{G}}^{\perp}$, in which:

$$\hat{\boldsymbol{G}}^{\parallel} = \frac{\boldsymbol{\kappa}(\boldsymbol{\kappa}\cdot\boldsymbol{G})}{\kappa^2} \qquad \text{and} \qquad \hat{\boldsymbol{G}}^{\perp} = \hat{\boldsymbol{G}} - \hat{\boldsymbol{G}}^{\parallel} = \hat{\boldsymbol{G}} - \frac{\boldsymbol{\kappa}(\boldsymbol{\kappa}\cdot\boldsymbol{G})}{\kappa^2};$$

written in components:

$$\hat{G}_j^{\parallel} = \frac{\kappa_j\kappa_k}{\kappa^2}\hat{G}_k \qquad \text{and} \qquad \hat{G}_j^{\perp} = P_{jk}\hat{G}_k = \left(\delta_{jk} - \frac{\kappa_j\kappa_k}{\kappa^2}\right)\hat{G}_k \tag{4.10}$$

where P_{jk} is the projection tensor.

4.4.2 The problem

The problem is modelled by the classical Navier-Stokes equations, under the hypothesis of incompressible Newtonian fluid with constant properties, written in differential form and with dimensionless quantities.

The velocity vector and pressure can be decomposed in average and floating component: $\boldsymbol{U}=<\boldsymbol{U}>+\boldsymbol{u}$ and $p_s=<p_s>+p$. Average velocity and pressure are then considered to be zero for the flow: $<\boldsymbol{U}>=0$ and $<p_s>=0$.

The Navier-Stokes equations for the fluctuating components are then:

$$\begin{cases} \nabla\cdot\boldsymbol{u} = 0 \\ \dfrac{\partial\boldsymbol{u}}{\partial t} + (\boldsymbol{u}\cdot\nabla)\boldsymbol{u} - \dfrac{1}{Re}\nabla^2\boldsymbol{u} + \nabla p = \boldsymbol{f} \end{cases}$$

where $\boldsymbol{u}=\boldsymbol{u}(x,y,z,t)$ is the floating component of the velocity vector (for the hypothesis of zero average velocity, it coincides with the velocity vector), $p=p(x,y,z,t)$ is the distribution of the floating component of pressure (it coincides with pressure), $\boldsymbol{f}=\boldsymbol{f}(x,y,z,t)$ is the forcing term per unit volume and Re is the Reynolds number.

For computational efficiency reasons, the nonlinear (convective) terms are better written in *divergence form* as $\nabla\cdot(\boldsymbol{u}\otimes\boldsymbol{u})$ rather than in the *convective form* previously used.

The equations are solved in a control volume $\mathcal{V}$ of cubical shape with dimension $\ell = 2\pi$, with right hand orthogonal cartesian reference system.

By projecting the velocity vector along the unit vectors of the reference system $\boldsymbol{u}$=$\{u,v,w\}$ it's possible to rewrite the Navier-Stokes equations in scalar form:

$$\begin{cases} \dfrac{\partial u}{\partial x} + \dfrac{\partial v}{\partial y} + \dfrac{\partial w}{\partial z} = 0 \\ \dfrac{\partial u}{\partial t} + \dfrac{\partial (uu)}{\partial x} + \dfrac{\partial (uv)}{\partial y} + \dfrac{\partial (uw)}{\partial z} = -\dfrac{\partial p}{\partial x} + \dfrac{1}{Re}\left(\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} + \dfrac{\partial^2 u}{\partial z^2}\right) + f_x \\ \dfrac{\partial v}{\partial t} + \dfrac{\partial (vu)}{\partial x} + \dfrac{\partial (vv)}{\partial y} + \dfrac{\partial (vw)}{\partial z} = -\dfrac{\partial p}{\partial y} + \dfrac{1}{Re}\left(\dfrac{\partial^2 v}{\partial x^2} + \dfrac{\partial^2 v}{\partial y^2} + \dfrac{\partial^2 v}{\partial z^2}\right) + f_y \\ \dfrac{\partial w}{\partial t} + \dfrac{\partial (wu)}{\partial x} + \dfrac{\partial (wv)}{\partial y} + \dfrac{\partial (ww)}{\partial z} = -\dfrac{\partial p}{\partial z} + \dfrac{1}{Re}\left(\dfrac{\partial^2 w}{\partial x^2} + \dfrac{\partial^2 w}{\partial y^2} + \dfrac{\partial^2 w}{\partial z^2}\right) + f_z \end{cases}$$

The three components of velocity and the pressure can be represented with the Fourier series thanks to the homogeneity of the problem, although introducing an artificial periodicity on the solution.

The Fourier series has $N+1$ terms in each direction, for a total of $(N+1)^3$ coefficients (or modes) of the series. The corresponding spatial discretisation has a step of $\Delta x = \frac{2\pi}{N}$ in the three directions. The unknown quantities, in the wavenumber domain, are the $(N+1)^3$ modal amplitudes $\hat{\boldsymbol{u}}_{\boldsymbol{\kappa}}(t)$; equally, in the physical domain the unknown quantities are the three velocity vector components $\boldsymbol{u}(\boldsymbol{x},t)$ in each of the $(N+1)^3$ points in an uniform grid with spacing Δx: there is a one-to-one mapping between amplitudes and velocities on the grid points.

By exploiting the definition of $\boldsymbol{\kappa}$ (Eq. (4.5)), and considering that the domain dimensions guarantee $\kappa_0 = 1$, it follows that:

$$\boldsymbol{\kappa} = \{l, m, n\} \qquad \text{with} \qquad l, m, n = 0 \dots N.$$

The solution is a velocity field which is homogeneous, but is also a physical quantity and as such mathematically is a real function; with the hypothesis of conjugated symmetry (Eq. (4.7)), it's possible to write: $\hat{\boldsymbol{u}}(\boldsymbol{\kappa},t) = \hat{\boldsymbol{u}}^*(-\boldsymbol{\kappa},t)$. The Hermitian property allows to eliminate the coefficient corresponding to negative modes, in a particular direction, from the unknown quantities, thus halving their number. Exploiting this property in the z direction, the wavenumber vector becomes:

$$\boldsymbol{\kappa} = \{l, m, n\} \qquad \text{with} \qquad l, m = 0 \dots N \quad \textit{and} \quad n = 0 \dots \frac{N}{2}$$

The spectral representation of velocity field and pressure is:

$$u(x,y,z,t) = \sum_{l=0}^{N}\sum_{m=0}^{N}\sum_{n=0}^{N/2} \hat{u}_{l,m,n}(t) e^{ilx_l} e^{imy_m} e^{inz_n}$$
$$v(x,y,z,t) = \sum_{l=0}^{N}\sum_{m=0}^{N}\sum_{n=0}^{N/2} \hat{v}_{l,m,n}(t) e^{ilx_l} e^{imy_m} e^{inz_n}$$
$$w(x,y,z,t) = \sum_{l=0}^{N}\sum_{m=0}^{N}\sum_{n=0}^{N/2} \hat{w}_{l,m,n}(t) e^{ilx_l} e^{imy_m} e^{inz_n}$$
$$p(x,y,z,t) = \sum_{l=0}^{N}\sum_{m=0}^{N}\sum_{n=0}^{N/2} \hat{p}_{l,m,n}(t) e^{ilx_l} e^{imy_m} e^{inz_n}$$

where $x_l = l\Delta x$, $y_m = m\Delta x$, $z_n = n\Delta x$; the time dependence appears only inside the series coefficients.

Applying the Fourier transform operator to the balance of momentum equations corresponds to the application of the Fourier operator $\mathcal{F}_\kappa\{\cdot\}$ (Eq. (4.4)) to each term of the equations, thanks to the property of linearity (Eq. (4.6)). The balance of momentum in the x direction is:

$$\mathcal{F}_\kappa\left\{\frac{\partial u}{\partial t}\right\} = \frac{d\hat{u}_{l,m,n}(t)}{dt}$$
$$\mathcal{F}_\kappa\left\{\frac{\partial(uu)}{\partial x} + \frac{\partial(vv)}{\partial y}\frac{\partial(ww)}{\partial z}\right\} = \widehat{HU}_{l,m,n}(t)$$
$$\mathcal{F}_\kappa\left\{-\frac{\partial p}{\partial x}\right\} = -il\hat{p}_{l,m,n}(t)$$
$$\mathcal{F}_\kappa\left\{\frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right)\right\} = -\frac{1}{Re}\kappa^2\hat{u}_{l,m,n}(t)$$
$$\mathcal{F}_\kappa\{f_x\} = \widehat{fx}_{l,m,n}(t)$$

with $\kappa^2 = l^2 + m^2 + n^2$.

By defining $\{\hat{u}_{l,m,n}(t), \hat{v}_{l,m,n}(t), \hat{w}_{l,m,n}(t)\} = \{\hat{u}_1, \hat{u}_2, \hat{u}_3\} = \{\hat{u}_j\}$, and similarly for the forcing and nonlinear term components, and $\boldsymbol{\kappa} = \{\kappa_j\}$, the generic component of the balance of momentum equation is:

$$\frac{d\hat{u}_j}{dt} + \widehat{HU}_j = -i\kappa_j\hat{p}_{l,m,n}(t) - \frac{1}{Re}\kappa^2\hat{u}_j + \widehat{fx}_j. \tag{4.11}$$

The continuity equation allows to express pressure as a function of the velocity vector; by applying the transform operator it becomes:

$$l\hat{u}_{l,m,n}(t) + m\hat{v}_{l,m,n}(t) + n\hat{w}_{l,m,n}(t) = 0 \quad \Rightarrow \quad \kappa_k\hat{u}_k = 0.$$

Multiplying each k-th component of the balance of momentum equations for the corresponding component of the wavenumber vector κ_k, adding and the considering the continuity equation ($\kappa_k \hat{u}_k = 0$) it's possible to obtain the Poisson equation for pressure in the wavenumber domain:

$$-i\kappa^2 \hat{p}_{l,m,n}(t) = \kappa_k \widehat{HU}_k - \kappa_k \widehat{fx}_k$$

by multiplying each term for $\frac{\kappa_j}{\kappa^2}$ the pressure is non-local and nonlinear:

$$-i\kappa_j \hat{p}_{l,m,n}(t) = \frac{\kappa_j \kappa_k}{\kappa^2} \widehat{HU}_k - \frac{\kappa_j \kappa_k}{\kappa^2} \widehat{fx}_k.$$

By substituting the pressure just obtained in the balance of momentum equation and grouping:

$$\frac{d\hat{u}_j}{dt} + \frac{1}{Re}\kappa^2 \hat{u}_j = -\left(\delta_{jk} - \frac{\kappa_j \kappa_k}{\kappa^2}\right) \widehat{HU}_k + \left(\delta_{jk} - \frac{\kappa_j \kappa_k}{\kappa^2}\right) \widehat{fx}_k$$

with the definition of projection tensor stated before:

$$\widehat{HU}_j^{\perp} = \left(\delta_{jk} - \frac{\kappa_j \kappa_k}{\kappa^2}\right) \widehat{HU}_k = P_{jk} \widehat{HU}_k$$
$$\widehat{fx}_j^{\perp} = \left(\delta_{jk} - \frac{\kappa_j \kappa_k}{\kappa^2}\right) \widehat{fx}_k = P_{jk} \widehat{fx}_k.$$

For each component of the balance of momentum equation, and for each wavenumber $\boldsymbol{\kappa} = \{l, m, n\}$, the Navier-Stokes equations in spectral representation are:

$$\begin{cases} \dfrac{d\hat{u}_{l,m,n}(t)}{dt} + \dfrac{1}{Re}\kappa^2 \hat{u}_{l,m,n}(t) = -\widehat{HU}^{\perp}_{l,m,n}(t) + \widehat{fx}^{\perp}_{l,m,n}(t) \\ \dfrac{d\hat{v}_{l,m,n}(t)}{dt} + \dfrac{1}{Re}\kappa^2 \hat{v}_{l,m,n}(t) = -\widehat{HV}^{\perp}_{l,m,n}(t) + \widehat{fy}^{\perp}_{l,m,n}(t) \\ \dfrac{d\hat{w}_{l,m,n}(t)}{dt} + \dfrac{1}{Re}\kappa^2 \hat{w}_{l,m,n}(t) = -\widehat{HW}^{\perp}_{l,m,n}(t) + \widehat{fz}^{\perp}_{l,m,n}(t) \end{cases}$$

4.4.3 Pseudo-spectral method

The *pseudo-spectral* approach originates from the *spectral* method, which consists in the application of Fourier transforms to the problem, thanks to the homogeneity of the solution. There are many advantages to this technique:

- exploiting the discrete Fourier transform to bring the problem in the wavenumber domain, the nature of the equations change from PDEs (partial differential equations) to ODEs (ordinary differential equations), much easier to solve with several numerical schemes;

- the Hermitian property allows to halve the unknown quantities, thus reducing the computational cost by a noticeable part;
- derivatives computation, using the derivation property of Fourier transform, is extremely accurate.

One drawback is the need to write the unknown quantities in the wavenumber domain through Fourier series and the calculation of the discrete coefficients. To do this, the best algorithm is the Fast Fourier Transform (FFT) published for the first time in 1965 by Cooley and Tukey [CT65](although earlier works existed on the subject), together with it's inverse version IFT (Inverse Fourier Transform). The basic idea behind this method is to decompose recursively a DFT of dimension $N = n_1 n_2$ in two smaller DFT of dimensions n_1 and n_2.

For the accuracy in floating-point arithmetic, the error is $\mathcal{O}(\varepsilon\ log_2 N)$ in the worst case (Gentleman and Sande [GS66]), and $\mathcal{O}(\varepsilon\sqrt{log_2 N})$ on average (Schatzman [Sch96]), compared to $\mathcal{O}(N^{3/2})$ and $\mathcal{O}(\varepsilon\sqrt{N})$ of the direct DFT implementation. For these values to hold it's necessary that certain trigonometric constants used by the algorithm (the *twiddle factors*) are computed correctly.

More significant is the fact that this algorithm manages to reduce the complexity of computing the discrete transform from $\mathcal{O}(N^2)$, which appears with the simple application of the DFT definition, to $\mathcal{O}(N\ logN)$, where N is the data size.

Another issue are the nonlinear terms. Considering for example the generic term $\widehat{HU}_j$:

$$\begin{aligned}
\widehat{HU}_j &= \mathcal{F}_\kappa \left\{ \frac{\partial (u_j u_k)}{\partial x_k} \right\} \\
&= i\kappa_k \mathcal{F}_\kappa \{u_j u_k\} \\
&= i\kappa_k \mathcal{F}_\kappa \left\{ \left(\sum_{\boldsymbol{\kappa}'} \hat{u}_j(\boldsymbol{\kappa}') e^{i\boldsymbol{\kappa}'\cdot\boldsymbol{x}} \right) \left(\sum_{\boldsymbol{\kappa}''} \hat{u}_k(\boldsymbol{\kappa}'') e^{i\boldsymbol{\kappa}''\cdot\boldsymbol{x}} \right) \right\} \\
&= i\kappa_k \sum_{\boldsymbol{\kappa}'} \sum_{\boldsymbol{\kappa}''} \hat{u}_j(\boldsymbol{\kappa}') \hat{u}_k(\boldsymbol{\kappa}'') \mathcal{F}_\kappa \left\{ e^{i(\boldsymbol{\kappa}'+\boldsymbol{\kappa}'')\cdot\boldsymbol{x}} \right\}.
\end{aligned}$$

By definition:

$$\mathcal{F}_\kappa \left\{ e^{i(\boldsymbol{\kappa}'+\boldsymbol{\kappa}'')\cdot\boldsymbol{x}} \right\} = \frac{1}{N^3} \sum_{\boldsymbol{x}} e^{i(\boldsymbol{\kappa}'+\boldsymbol{\kappa}'')\cdot\boldsymbol{x}} e^{-i\boldsymbol{\kappa}\cdot\boldsymbol{x}} = \delta(\boldsymbol{\kappa} - (\boldsymbol{\kappa}' + \boldsymbol{\kappa}'')),$$

exploiting the property of the Dirac delta function the result is:

$$\widehat{HU}_j = i\kappa_k \sum_{\boldsymbol{\kappa}'} \hat{u}_j(\boldsymbol{\kappa}') \hat{u}_k(\boldsymbol{\kappa} - \boldsymbol{\kappa}')$$

which corresponds to a discrete convolution of function $\hat{u}_j$ and $\hat{u}_k$ in the wavenumber domain. This is the issue: to compute this convolution it's necessary to resolve six convolution products of the velocity vector components, thus requiring the high amount of $\mathcal{O}(N^6)$ operations.

The solution comes from the convolution theorem (Eq. (4.9)), which states that a convolution in the wavenumber domain corresponds to a product in the physical domain.

To compute the nonlinear terms, the procedure is the following:

1. transform the velocity components from the wavenumber domain to the physical domain with IFT:

$$\{\hat{u}_{l,m,n}(t), \hat{v}_{l,m,n}(t), \hat{w}_{l,m,n}(t)\} \quad \Rightarrow \quad \{u(\boldsymbol{x},t), v(\boldsymbol{x},t), w(\boldsymbol{x},t)\}$$

2. compute the six mixed products in the physical domain:

$$\begin{pmatrix} u(\boldsymbol{x},t)u(\boldsymbol{x},t) & u(\boldsymbol{x},t)v(\boldsymbol{x},t) & u(\boldsymbol{x},t)w(\boldsymbol{x},t) \\ & v(\boldsymbol{x},t)v(\boldsymbol{x},t) & v(\boldsymbol{x},t)w(\boldsymbol{x},t) \\ & & w(\boldsymbol{x},t)w(\boldsymbol{x},t) \end{pmatrix}$$

3. transform the products just obtained back to the wavenumber domain with FFT:

$$\begin{pmatrix} \widehat{uu}_{l,m,n}(t) & \widehat{uv}_{l,m,n}(t) & \widehat{uw}_{l,m,n}(t) \\ & \widehat{vv}_{l,m,n}(t) & \widehat{vw}_{l,m,n}(t) \\ & & \widehat{ww}_{l,m,n}(t) \end{pmatrix}$$

The derivation property allows to write:

$$\widehat{HU}_j = \mathcal{F}_\kappa \left\{ \frac{\partial(u_j u_k)}{\partial x_k} \right\} = i\kappa_k \mathcal{F}_\kappa\{u_j u_k\} = i\kappa_k \widehat{u_j u_k},$$

then expliciting the nonlinear terms gives:

$$\begin{aligned} \widehat{HU}_{l,m,n}(t) &= il\widehat{uu}_{l,m,n}(t) + im\widehat{uv}_{l,m,n}(t) + in\widehat{uw}_{l,m,n}(t) \\ \widehat{HV}_{l,m,n}(t) &= il\widehat{uv}_{l,m,n}(t) + im\widehat{vv}_{l,m,n}(t) + in\widehat{vw}_{l,m,n}(t) \\ \widehat{HW}_{l,m,n}(t) &= il\widehat{uw}_{l,m,n}(t) + im\widehat{vw}_{l,m,n}(t) + in\widehat{ww}_{l,m,n}(t) \end{aligned} \tag{4.12}$$

This is the *pseudo-spectral* method, it allows to compute the nonlinear terms in a much more efficient way, requiring only $\mathcal{O}(N^3\ logN)$ with the 3D FFT algorithm.

There's another aspect in this method that requires particular attention: aliasing. When executing the products in the physical domain on data characterised by a discrete limited band N, and then transforming them in the wavenumbers domain, it's possible to obtain results characterised by

harmonics associated with wavenumbers outside the allowed range (band). These terms are erroneously repositioned inside the band of interest, translated backward by the quantity N, thus generating errors on the solution and instability.

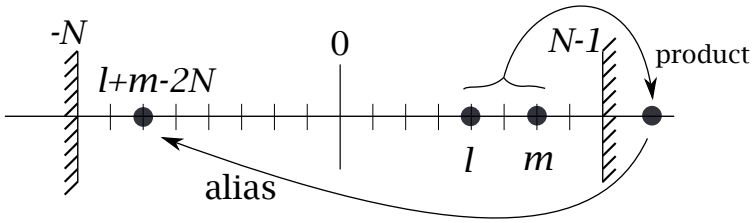


Figure 4.5: Aliasing error resulting from products. (Adapted from [Rog81])

This error can't be removed after the computation, but it's necessary to apply an *anti-aliasing* technique; some of them are: phase shift, truncating (2/3-rule) and zero padding (3/2-rule, used in this case). The 3/2-rule is applied before transforming the solution to the physical space, and consists in increasing the number of modes to at least 1.5 times the default number ($2N$), and setting to *0* the newly introduced modes. This technique guarantees that the products computed with the pseudo-spectral method corresponds to the ones obtained by the convolution products in the spectral method.

Furthermore, the FFT algorithm requires vectors with dimensions that are powers of 2, so the expanded vectors will have dimensions of at least $3N$ in non-Hermitian directions and $3/2N$ in Hermitian directions.

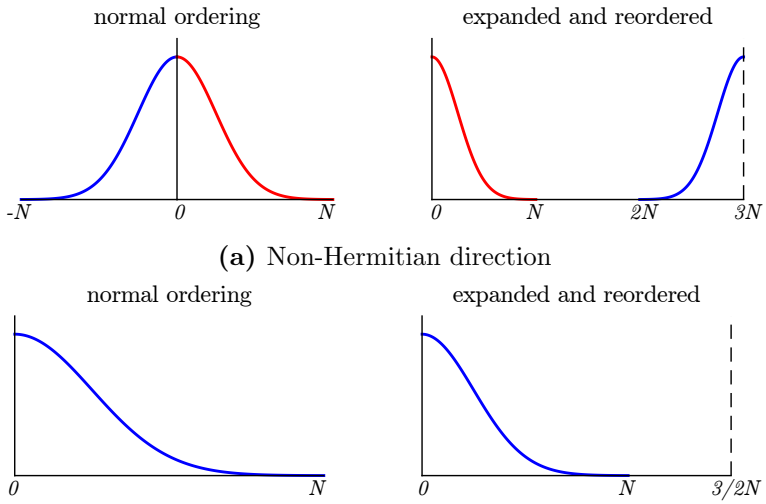
An example is illustrated in Fig. 4.6: the original data is copied in a bigger array where the are modes outside the original domain are set to zero. Not only there is an expansion (for de-aliasing purposes), but also a reordering is needed to comply with the FFT library's convention, as will be explained in Section 5.7.

Once the non-linear products are executed and transformed back in the wavenumbers domain, the added modes containing spurious terms are not considered, while the original modes obtained are correctly without the aliasing error.

It's important to consider that when $\kappa^2 = 0$, that is $l = m = n = 0$, the equations become singular: this combination of wavenumbers defines the average values of the velocity vector components that, in the fluctuating velocity component formulation used in this case, are zero and must not be computed [QL01].

Forcing

It's necessary to add a forcing term in the balance of momentum equations in this case since the homogeneous isotropic flow needs an energy supply



(a) Non-Hermitian direction

(b) Hermitian direction

Figure 4.6: Expansion and reordering of the coefficients' array.

in order to sustain itself and reach a statistically stationary state. It's an artificial term so it's defined in the wavenumber domain; the energy introduction happens on large scale structures, that is for wavenumber $|\boldsymbol{\kappa}| \leq \kappa_f$, with κ_f sufficiently small.

In this case, the forcing term defined by Caughey, Lamorgese and Pope [LCP05] has been used:

$$\widehat{fx_j}^{\perp}(\boldsymbol{\kappa}, t) = \begin{cases} \dfrac{P}{2k_f(t)} \hat{u}_j(\boldsymbol{\kappa}, t) = f_k \hat{u}_j(\boldsymbol{\kappa}, t) & \text{if } |\boldsymbol{\kappa}| \leq \kappa_f \\ 0 & \text{if } |\boldsymbol{\kappa}| > \kappa_f \end{cases} \tag{4.13}$$

where k_f is the kinetic energy contained in those wavenumbers, and P is the average dissipation in the statistically stationary state. It's necessary to define the shell amplitude on which the forcing term operates, in this work it is $\kappa_f/\kappa_0 = 3$, and the value of P, which is 1 in this case.

The parameter f_κ varies depending on the solution while the simulation proceeds, in order to guarantee a constant energy introduction.

4.4.4 Time discretisation

After the spatial discretisation, to solve the Navier-Stokes equations is necessary to apply a time discretisation, since they are evolutionary differential equations.

The initial time is $t_0 = 0$, the simulations time is T and the time domain is divided in N_t intervals, so that:

$$\Delta t = \frac{T}{N_t} \quad \Rightarrow \quad t_k = k\Delta T \quad \text{with} \quad k = 0 \dots N_t,$$

where N_t (and therefore Δt) is chosen based on the time discretisation technique in order to guarantee good numerical stability properties.

Time discretisation techniques can be divided in two main categories: *explicit* and *implicit*. Considering the model problem $\dot{x}(t) = s(x(t), t)$, it can be approximated with a generic multi-step linear method as:

$$x^{(k+1)} = \sum_{i=0}^{s} a_i x^{(k+1)} + \Delta t \sum_{i=-1}^{s} b_i f(x^{(k-i)}, t^{(k-i)}).$$

If the solution at the present step depends from the forcing term at the present step, the approximation is called *implicit*; if it doesn't, it's called *explicit*. Implicit discretisation techniques are more expensive from a computational point of view since they require to solve a linear system, but they are unconditionally stable. Explicit methods are easier to solve but are conditionally stable under the so called *CFL condition*: if the forcing term can be expressed as $f(x(t), t) = ax(t)$, with a convection parameter, the explicit method is numerically stable if:

$$CFL = a\frac{\Delta t}{\Delta x} < 1.$$

The Navier-Stokes equations for the generic component j and for each wavenumber $\boldsymbol{\kappa} = \{l, m, n\}$ are:

$$\frac{d\hat{u}_j}{dt} + \frac{1}{Re}\kappa^2 \hat{u}_j = -P_{jk}\widehat{HU}_k + \widehat{fx}_j^{\perp} \qquad \text{with} \qquad \widehat{HU}_j = i\kappa_k \widehat{u_j u_k},$$

By using an implicit technique, the resulting equations are coupled due to the presence of nonlinear and forcing terms (if it depends from the solution like in this case). The *exact treatment* of viscous terms can be used if a higher level of accuracy is desired. Another method is the *semi-implicit approximation*, which discretises implicitly the viscous terms but explicitly the non-linear and forcing terms to obtain decoupled equations which can be solved separately.

Exact treatment

The aim of the *exact treatment* is to compute in an exact way the viscous and forcing terms. The errors on the solution derive only from the numerical discretisation and the approximation of the non-linear terms in explicit way. By using the definition of forcing term (Eq. (4.13)), the x component of the balance of momentum is:

$$\left(\frac{d}{dt} + \frac{1}{Re}\kappa^2 - f_\kappa\right)\hat{u} = -\widehat{HU}_\perp,$$

where $\hat{u} = \hat{u}_{l,m,n}(t)$, $\widehat{HU}_\perp = \widehat{HU}^\perp_{l,m,n}(t)$ and $f_\kappa \hat{u} = \widehat{fx}^\perp_{l,m,n}(t)$.

By introducing the operator $F = e^{\left(\frac{1}{Re}\kappa^2 - f_\kappa\right)t}$, the problem can be rewritten as:

$$F^{-1}(F\hat{u})_{/t} = -\widehat{HU}_\perp,$$

which is an exact representation of the starting problem.

By making explicit and discretising the evolutionary term, it follows:

$$(F\hat{u})_{/t} = \frac{(F\hat{u})^{k+1} - (F\hat{u})^k}{\Delta t} = e^{\left(\frac{1}{Re}\kappa^2 - f_\kappa\right)t^{(k)}} \; \frac{e^{\left(\frac{1}{Re}\kappa^2 - f_\kappa\right)\Delta t}\hat{u}^{(k+1)} - \hat{u}^{(k)}}{\Delta t}$$

which, substituted in the previous equation, gives:

$$F^{-1}(F\hat{u})_{/t} = \frac{e^{\left(\frac{1}{Re}\kappa^2 - f_\kappa\right)\Delta t}\hat{u}^{(k+1)} - \hat{u}^{(k)}}{\Delta t} = -a\widehat{HU}_\perp^{(k)} - b\widehat{HU}_\perp^{(k-1)},$$

where a and b are used to choose the particular explicit method.

At the end, the equations to solve the problem with the exact representation of viscous and forcing terms are:

$$\begin{aligned}
\hat{u}^{(k+1)} &= e^{-\left(\frac{1}{Re}\kappa^2 - f_\kappa\right)\Delta t}\left[\hat{u}^{(k)} + \Delta t\left(-a\widehat{HU}_\perp^{(k)} - b\widehat{HU}_\perp^{(k-1)}\right)\right] \\
\hat{v}^{(k+1)} &= e^{-\left(\frac{1}{Re}\kappa^2 - f_\kappa\right)\Delta t}\left[\hat{v}^{(k)} + \Delta t\left(-a\widehat{HV}_\perp^{(k)} - b\widehat{HV}_\perp^{(k-1)}\right)\right] \\
\hat{w}^{(k+1)} &= e^{-\left(\frac{1}{Re}\kappa^2 - f_\kappa\right)\Delta t}\left[\hat{w}^{(k)} + \Delta t\left(-a\widehat{HW}_\perp^{(k)} - b\widehat{HW}_\perp^{(k-1)}\right)\right].
\end{aligned} \tag{4.14}$$

Semi-implicit approximation

This method requires separate treatment of the right hand side terms. Considering the model problem:

$$\frac{dy}{dt} = f_1 + f_2,$$

where $f_1 = f_1(y(t), t)$ will be treated in explicit mode, and $f_2 = f_2(y(t), t)$ in implicit. A general semi-implicit discretisation in two steps can be written as:

$$y^{(k+1)} = y^{(k)} + \Delta t\left(af_1^{(k)} + bf_1^{(k-1)} + cf_2^{(k+1)} + df_2^{(k)}\right),$$

where $\{a$,b$\}$ defines the implicit method and $\{c$,d$\}$ defines the implicit one.

Consider the x component of the momentum balance equation in spectral representation:

$$\frac{d\hat{u}}{dt} = \left(-\widehat{HU}_\perp + \widehat{fx}_\perp\right) + \left(-\frac{1}{Re}\kappa^2\hat{u}\right),$$

where $\hat{u} = \hat{u}_{l,m,n}(t)$, $\widehat{HU}_\perp = \widehat{HU}^\perp_{l,m,n}(t)$ and $\widehat{fx}_\perp = \widehat{fx}^\perp_{l,m,n}(t)$.

The first bracket on the right hand side is treated in explicit mode while the second will be treated implicitly, in a similar way to the model problem. Discretising the equation gives:

$$\hat{u}^{(k+1)} = \hat{u}^{(k)} + \Delta t \Bigg[a \left(-\widehat{HU}_\perp^{(k)} + \widehat{fx}_\perp^{\,(k)} \right) + b \left(-\widehat{HU}_\perp^{(k-1)} + \widehat{fx}_\perp^{\,(k-1)} \right) \Bigg) + \dots + c \left(-\frac{1}{Re} \kappa^2 \hat{u}^{(k+1)} \right) + d \left(-\frac{1}{Re} \kappa^2 \hat{u}^{(k)} \right) \Bigg].$$

By developing the equation and regrouping the velocity component at the $k+1$ step in the left side, the equations for the semi-implicit approximated method are:

$$\begin{aligned}
\hat{u}^{(k+1)} &= \frac{1}{1 + c\dfrac{\Delta t}{Re}\kappa^2} \Bigg\{ \left(1 - d\frac{\Delta t}{Re}\kappa^2 \right) \hat{u}^{(k)} + \dots \\
&\quad + \Delta t \left[a \left(-\widehat{HU}_\perp^{(k)} + \widehat{fx}_\perp^{\,(k)} \right) + b \left(-\widehat{HU}_\perp^{(k-1)} + \widehat{fx}_\perp^{\,(k-1)} \right) \right] \Bigg\} \\
\hat{v}^{(k+1)} &= \frac{1}{1 + c\dfrac{\Delta t}{Re}\kappa^2} \Bigg\{ \left(1 - d\frac{\Delta t}{Re}\kappa^2 \right) \hat{v}^{(k)} + \dots \\
&\quad + \Delta t \left[a \left(-\widehat{HV}_\perp^{(k)} + \widehat{fy}_\perp^{\,(k)} \right) + b \left(-\widehat{HV}_\perp^{(k-1)} + \widehat{fy}_\perp^{\,(k-1)} \right) \right] \Bigg\} \\
\hat{w}^{(k+1)} &= \frac{1}{1 + c\dfrac{\Delta t}{Re}\kappa^2} \Bigg\{ \left(1 - d\frac{\Delta t}{Re}\kappa^2 \right) \hat{w}^{(k)} + \dots \\
&\quad + \Delta t \left[a \left(-\widehat{HV}_\perp^{(k)} + \widehat{fz}_\perp^{\,(k)} \right) + b \left(-\widehat{HW}_\perp^{(k-1)} + \widehat{fz}_\perp^{\,(k-1)} \right) \right] \Bigg\}
\end{aligned} \tag{4.15}$$

Whether the exact or approximate treatment of viscous and forcing term is chosen, the time discretisation technique to be used is decided by setting the coefficients a and b; two options are:

- *forward Euler*: explicit linear one-step multi-step method with first order accuracy, with coefficients: *a=1*, *b=0*;
- *Adams–Bashforth 2*: explicit linear two-step multi-step method with second order accuracy, with coefficients: *a=1.5*, *b=-0.5*;

For the approximate method, a good choice is the *Crank-Nicolson* method, an implicit two-step second order method; it's imposed with coefficients: *c=0.5*, *d=0.5*.

4.4.5 Boundary and initial conditions

A well-posed problem needs boundary and initial conditions to be solved. For this case, homogeneous isotropic turbulence, with the velocity field represented with Fourier series, the best boundary condition is of periodic type. The solution at each point of each face of the surface which delimits the cubical domain $\mathcal{V}$ is equal to the corresponding point on the opposing face:

$$\begin{aligned}\boldsymbol{u}(0,y,z,t) &= \boldsymbol{u}(\ell,y,z,t)\\ \boldsymbol{u}(x,0,z,t) &= \boldsymbol{u}(x,\ell,z,t)\\ \boldsymbol{u}(x,y,0,t) &= \boldsymbol{u}(x,y,\ell,t).\end{aligned}$$

With this condition, the detailed specification of the highly chaotic motion at boundaries can be avoided, and it's consistent with the hypothesis of homogeneity and isotropy.

Because it's an *artificial* boundary condition, there's an undesired effect on the solution; to reduce this effect, which originates from the finite dimensions of the control volume, and obtain a valid solution it's necessary to increase the domain size ℓ as explained in Section 4.3.1

The initial conditions for a DNS can't be obtained from experimental data, since they are not sufficiently complete (having only some statistical data) and contains several sources of error. Good initial conditions can be created by following the divergence-free condition and are characterised by a specific energy spectrum, defined as:

$$E(\boldsymbol{\kappa},t) = \mathcal{F}_\kappa\{\hat{\boldsymbol{u}}^*(\boldsymbol{\kappa},t)\cdot\hat{\boldsymbol{u}}(\boldsymbol{\kappa},t)\}.$$

One way to construct the initial conditions is the following procedure:

1. define an initial energy spectrum: Caughey, Lamorgese and Pope [LCP05] suggests the following energy spectrum:

$$E(\boldsymbol{\kappa},0) = \frac{9}{11}\frac{1}{\kappa_f}\times\begin{cases}\left(\dfrac{\kappa}{\kappa_f}\right)^2 & \text{if } |\boldsymbol{\kappa}|\le\kappa_f\\ \left(\dfrac{\kappa}{\kappa_f}\right)^{-\frac{5}{3}} & \text{if } |\boldsymbol{\kappa}|>\kappa_f\end{cases}$$

   where κ_f is small and represents the maximum wavenumber at which the forcing term is applied;

2. for each wavenumber vector $\boldsymbol{\kappa}=\{l,m,n\}$, define two velocity component randomly:

$$\begin{aligned}\hat{u}_{l,m,n}(0) &= \sqrt{\frac{E(\boldsymbol{\kappa},0)}{4\pi\kappa^2}}e^{i2\pi\cdot\mathrm{rand}}\\ \hat{v}_{l,m,n}(0) &= \sqrt{\frac{E(\boldsymbol{\kappa},0)}{4\pi\kappa^2}}e^{i2\pi\cdot\mathrm{rand}}\end{aligned}\tag{4.16}$$

where *rand* generates random numbers in the interval (0,1);

3. for each wavenumber vector $\boldsymbol{\kappa} = \{l, m, n\}$, impose the divergence-free condition to obtain the third velocity component:

$$\hat{w}_{l,m,n}(0) = -\frac{il\hat{u}_{l,m,n}(0) + im\hat{v}_{l,m,n}(0)}{in}; \tag{4.17}$$

An alternative to this method are the initial conditions of the *Taylor-Green* vortex flow, defined in physical space; it's analysed in Chapter 6.

It's important to highlight that these kinds of initial conditions are not realistic, it's energy spectrum is very different from the one obtained at converge: it will take more time to reach a statistically stationary state. To improve converge speed, the solution from a previous simulation of turbulent flow without forcing term should be used as initial condition: the velocity field is divergence-free by construction and its energy spectrum is more similar to the final one, thus improving convergence speed.

4.4.6 Degrees of freedom for the problem

Not all physical and numerical parameters can be chosen freely.

To adapt the dimensionless equations of the problem it's necessary to define a *turbulent Reynolds number*:

$$Re_t = \frac{\mathcal{L}\,\mathcal{U}}{\nu}$$

where ν is the kinematic viscosity of the fluid; the characteristic length $\mathcal{L}$ and velocity $\mathcal{U}$ scales are determined by the choice of the wavenumber κ_f.

The dimensionless coefficient κ_f/κ_0 is chosen as degree of freedom, from which it's possible to obtain κ_f knowing that the dominion's dimensions guarantee that $\kappa_0 = 1$. Once κ_f is known, the length and velocity scales are defined as:

$$\mathcal{L} = \kappa_f^{-1} \qquad \text{and} \qquad \mathcal{U} = \left(\frac{P}{\kappa_f}\right)^{1/3}$$

where P=<ϵ>=1 is the average dissipation in statistically stationary conditions.

The Reynolds number is defined as:

$$Re = \left(\frac{\kappa_f}{\kappa_0}\right)^{4/3} Re_t.$$

From the definition of characteristics length and velocity, the Reynolds number can be written as:

$$Re = \left(\frac{\kappa_d}{\kappa_0}\right)^{4/3},$$

which defines a measure of the dissipative microscales of Kolmogrov η and thus the maximum wavenumber κ_d since $\kappa_d = \eta^{-1}$.

In order to simply the application of the Hermitian property of the solution, in the developed program it was decided to index the Fourier coefficients in the following way: $\{-N, -N+1, \ldots, -1, 0, 1, \ldots, N\}$. The effective number of modes used to represent the solution is $2N + 1$ in non-Hermitian directions, and $N + 1$ in the Hermitian direction.

The time discretisation step Δt has to be chosen carefully. Similarly to the CFL condition for an explicit method, the *Courant number* is introduced like it was previously discussed:

$$C = \frac{k^{1/2}\Delta t}{\Delta x},$$

where k is the turbulent kinetic energy of the flow. The spatial step Δx is determined by the purpose to achieve maximum accuracy with the available computational resources. To guarantee numerical stability it's fundamental that the time step Δt gives $C < C_{max}$, where C_{max} depends from the discretisation method used (generally is much smaller than 1).

The total simulation time T_{max}, for a turbulent forced flow, has to guarantee the statistically stationary condition, that is the flow has constant (and non-zero) energy.

The parameters Re and N depend from each other. Once the the turbulent Reynolds number is fixed (thus deciding the kind of flow to study), the choice of number of modes comes from the interpolation of data available in literature [LCP05] for non-forced flows; if the flow is forced, with the same number of modes the Re has to be at least doubled.

5. The computer code

5.1 Organisation of the program

As previously stated, the program developed for this thesis is written in the CUDA C language. It's composed of three files:

- `cuNS.cu`: it contains the bulk of the program, that is the memory allocations for host and device memory, the implementation of two types of initial conditions, the time loop where the solution is iteratively computed and some tools for the post-processing of the results (validation and data export);
- `cuNS_kernels.cu`: it contains the functions that runs on the GPU (kernels), which includes the mathematical formulas explained before in chapter 4;
- `defines.h`: is a header file recalled by `cuNS.cu` and `cuNS_kernels.cu`, and it contains the definitions of the data structures used in the program, along with other macros.

5.2 The program

In the following section, the most important parts of the program will be explained.

5.2.1 Libraries

It's important to include the CUDA libraries: `cuda.h` and `cuda_runtime.h`; while it's not really needed if compiling with `nvcc` (because it automatically takes care of including the required headers), it is needed if the host code contains API calls and is compiled by the host compiler or the code uses some CUDA built-ins.

To handle complex numbers, the program needs the `cuComplex.h` library. This CUDA library defines the `cuDoubleComplex` type, which in reality is a simple redefinition of the `double2` type, which is a C vector data type with interleaved real and imaginary components `.x` and `.y`; the same thing is valid for single precision, as `cuComplex` is a redefinition of the `float2` type. Other than this, it contains several functions that operates on complex numbers like modulus, multiplication, division, fused multiply-add and so on, in both single and double precision. On the host code, complex

numbers are handled with their separate parts, so there's no need for the the ANSI C `complex.h` library.

The CUDA toolkit provides a fast mathematical library: with `#include "math.h"`, the GPU has access to high performance mathematical routines optimised for the architecture.

Obviously, this program revolves around Fourier transforms, so it's necessary to include the `cufft.h` library.

5.2.2 Important macros

The macros included in `defines.h` are fundamental for the functioning of the program and are used by both the main and the kernel's file. The key ones are:

- `N`: number of grid points (or Fourier modes) in x and y **half** directions for the velocity terms; the modes in these directions are $2 * N$;
- `Nz`: number of grid points (or Fourier modes) in the z **half** direction for the velocity terms;
- `N_ROWs_R`: $2 * N + 1$, number of rows (x-direction) for the physical domain velocity matrix;
- `N_ROWs_C`: $2 * N + 1$, number of rows (x-direction) for the wavenumbers domain velocity matrix;
- `N_COLs_R`: $2 * N + 1$, number of columns (y-direction) for the physical domain velocity matrix;
- `N_COLs_C`: $2 * N + 1$ in 3D or $N + 1$ in 2D, number of columns (y-direction) for the physical domain velocity matrix;
- `N_DEPs_R`: $2 * Nz + 1$ in 3D or 1 in 2D, level of depth (z-direction) for the physical domain velocity matrix;
- `N_DEPs_C`: $Nz + 1$ in 3D or 1 in 2D, level of depth (z-direction) for the wavenumbers domain velocity matrix;
- `DOM_SIZE_R`: $Nrows_R * Ncols_R * Ndeps_R = (2 * N + 1)^2(2 * Nz + 1)$, domain size, in number of grid points, of the real velocity data;
- `DOM_SIZE_C`: $Nrows_C * Ncols_C * Ndeps_C = (2 * N + 1)^2(Nz + 1)$, domain size, in number of Fourier modes (or series coefficients), of the complex velocity data;
- `N_EXP`: number of grid points (or Fourier modes) in x and y directions for the expanded non-linear terms;

- `Nz_EXP`: number of grid points (or Fourier modes) in the z direction for the expanded non-linear terms;
- `N_ROWs_EXP_C`: N_{exp}, number of rows for the wavenumbers domain non-linear terms matrix;
- `N_COLs_EXP_C`: N_{exp} in 3D or $N_{exp}+1$ in 2D, number of columns for the wavenumbers domain non-linear terms matrix;
- `N_DEPs_EXP_C`: $Nz_{exp}/2+1$, level of depth for the wavenumbers domain non-linear terms matrix;
- `EXP_DOM_SIZE_R`: domain size, in number of grid points, of the real non-linear terms data: $Nrows_{expR} * Ncols_{expR} * Ndeps_{expR} = N_{exp}^2(Nz_{exp})$.
- `EXP_DOM_SIZE_C`: domain size, in number of Fourier modes, of the complex non-linear terms data: $Nrows_{expC} * Ncols_{expC} * Ndeps_{expC} = N_{exp}^2(Nz_{exp}/2+1)$.

To insert a different calculation directly in a macro a whether the problem is 2D or 3D, the ANSI C `inline if (?:)` ternary operator is used. It means "*condition ? value_if_true : value_if_false*", and it's a shorthand for the *if-then-else* construct; for example, a macro using this construct can be written as:

```
#define N_COLs_C   ( (Nz==1) ? (N+1) : (2*N+1) )
```

5.2.3 Data structures

The definition of the data structures is included in `defines.h`, since they are used by both the main file and the kernels' file. There are five main data structures used by the program:

- ```
  struct velocityReal{ double u[2*DOM_SIZE_C],
                               v[2*DOM_SIZE_C],
                               w[2*DOM_SIZE_C];}
  ```

- ```
  struct velocityComp{ cuDoubleComplex u[DOM_SIZE_C],
                                       v[DOM_SIZE_C],
                                       w[DOM_SIZE_C];}
  ```

- ```
  struct nonLinearReal{ double uu[2*EXP_DOM_SIZE_C],
                                vv[2*EXP_DOM_SIZE_C],
                                ww[2*EXP_DOM_SIZE_C],
                                uv[2*EXP_DOM_SIZE_C],
                                uw[2*EXP_DOM_SIZE_C],
                                vw[2*EXP_DOM_SIZE_C];}
  ```
  ```

- `struct nonLinearComp{ cuDoubleComplex uu[EXP_DOM_SIZE_C], vv[EXP_DOM_SIZE_C], ww[EXP_DOM_SIZE_C], uv[EXP_DOM_SIZE_C], uw[EXP_DOM_SIZE_C], vw[EXP_DOM_SIZE_C];}`

- `struct rhsComp{ cuDoubleComplex rhsOldu[DOM_SIZE_C], rhsOldv[DOM_SIZE_C], rhsOldw[DOM_SIZE_C];}`

The real velocity structure and the real non-linear terms array contains array fields with dimensions that are double the ones in the complex structure due to the in-place transform and padding, more details on this in Section 5.2.4. To define these two structure in order to use them for out-of-place transforms, the respective fields must have dimensions `DOM_SIZE_R` and `EXP_DOM_SIZE_R`.

Each of these structures have been organised as a *structure of arrays* (SoA), opposed to the *array of structures* (AoS) approach. As evident from Fig. 5.1, if the GPU performs an operation only on the x field, there's a waste of 50% of the bandwidth with the AoS approach. Because there's no interleaving of elements of the same field in the SoA approach, the GPU obtains coalesced memory access and can achieve a more efficient global memory utilisation.

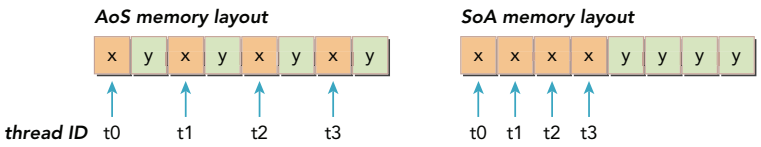


Figure 5.1: Memory layout of the two approaches.[CGM14]

Another important variable used by the program is an array formed by kinetic energy contributions from each wavenumber, for the purpose of using the kinetic energy in the forcing terms as explained in Eq. (4.13); the array has `DOM_SIZE_C` elements.

For DNS solvers, and in general for scientific computations, it's imperative the use of double precision floating point variables. The penalty for this added precision is a doubling of the memory size (8 bytes vs 4 bytes for a `float`).

The velocity structures contains the three components of the velocity vector; the non-linear structures contains the six cross-mixed products (Section 4.4.3) of the velocity vector components; the `rhsComp` structure contains the right hand side terms of the equations that are, in a time discretisation algorithm, the terms computed in the previous iteration.

From a memory occupancy point of view, a single element of the complex non-linear terms structure occupies 96 bytes (6 components × (8+8) bytes each), while each element of the complex velocity structure occupies 48 bytes.

5.2.4 Fast Fourier Transforms

Fourier transforms play a pivotal role in a pseudo-spectral DNS solver, as explained in Section 4.4.3. They take most of the computational time, hence it's important to employ a very efficient algorithm. Nvidia develops a proprietary library called *cuFFT* [Nvi15b] for this task; it's a highly optimised library modeled after the *Fastest Fourier Transform in the West* (FFTW) library [FJ14a], which is one of the most popular and efficient CPU-based FFT libraries.

This library uses a two-step procedure to execute FFTs. In the first stage there's a configuration mechanism called *plan* that uses internal building blocks to optimise the transform for the given configuration and the available GPU hardware. First, a plan variable needs to be declared with `cufftHandle plan` (this handle type is just a redefine of an `unsigned int`), then the command for creating a 3D plan of dimensions $n_x \times n_y \times n_z$ in cuFFT is:

```
cufftResult cufftPlan3d(cufftHandle *plan, int nx, int ny, int nz,
    cufftType type);
```

It returns a pointer to a generated plan of the specified type. The transform type can be one of six choices: complex to complex, real to complex or complex to real, each of these available also in a double precision version. Since in this work the double precision variables are used, the two types employed are `CUFFT_D2Z`(double precision real-to-complex) for the FFTs and `CUFFT_Z2D`(double precision complex-to-real) for the IFTs.

The second stage in the cuFFT procedure is the actual *execution* of the transforms, which follows the previously determined plan of execution; the command to execute a real valued double precision real-to-complex FFT is:

```
cufftResult cufftExecD2Z(cufftHandle plan, cufftComplexReal *in_data,
    cufftDoubleComplex *out_data);
```

The `cufftDoubleReal` and `cufftDoubleComplex` types are a simple redefinition of the `cuDoubleReal` and `cuDoubleComplex` types.

With the previous command the programmer can chose to execute an *in-place* or *out-of-place* transform. In the out-of-place way, the input and output data reside in different memory locations and don't overlap, meaning that they are in fact different variables. In the in-place transform, however, the input and output data share the same memory location, and one is simply a cast of the other in a different variable type. This effectively halves the required memory since there's no need for a separate output array, but

also reduces the performances of the transforms; the out-of-place mode is faster since uses more efficient kernels. Performance implications of this execution choice are discussed in Chapter 6.

The advantage of the two-stage approach is that the plan is created only once, and the library retains the state needed to execute the plan multiple times without needing a recalculation of the configuration. This model works well for cuFFT because different kinds of FFTs require different thread configurations and GPU resources, and the plan interface provide a simple way of reusing configurations.

Worksize

During plan execution, cuFFT requires a work area for temporary storage of intermediate results; this information can be obtained, once plan generation is done, with the function:

```
cufftResult cufftGetSize(cufftHandle plan, size_t *workSize);
```

which returns the bytes of memory required by the plan in the `workSize` variable. Some problems require much more storage than others, in particular powers of 2 are very efficient in terms of temporary storage.

Some measurements of plan storage requirements are gathered in Table 5.1; the velocity array has odd dimensions which practically are never powers of 3,5 or 7, so the FFTs executed on it will be very inefficient and the plans occupy a sizeable portion of memory.

N_{dir}	N_{exp}	expanded C2R	expanded R2C	C2R	R2C
21	32	0.49 MB	0.51 MB	2 MB	2.06 MB
42	64	4.52 MB	4.63 MB	16 MB	16.25 MB
85	128	37.48 MB	37.92 MB	128 MB	129 MB
169	256	299.87 MB	301.63 MB	1024 MB	1028 MB
341	512	1024 MB	1028 MB	2555 MB	2555 MB

Table 5.1: Memory needed for the plans used by the cuFFT library's functions

Sizes

The cuFFT library employs the Cooley-Tukey algorithm [CT65] to reduce the number of required operations to optimise the performance of particular transform sizes. This algorithm expresses the DFT matrix as a product of sparse building block matrices. The cuFFT library implements the radix-2, radix-3, radix-5, radix-7 building blocks, hence the performance of any size that can be factored as $2^a \times 3^b \times 5^c \times 7^d$ is optimised.

Moreover, it's better to restrict the size along each dimensions to use fewer distinct prime factors: a transform of size 2^n or 3^n will usually be faster than one of size $2^i \times 3^j$ even if the latter is slightly smaller.

In general a smaller prime factor leads to better performances: the powers of two are the fastest, as can be seen in Fig. 5.2.

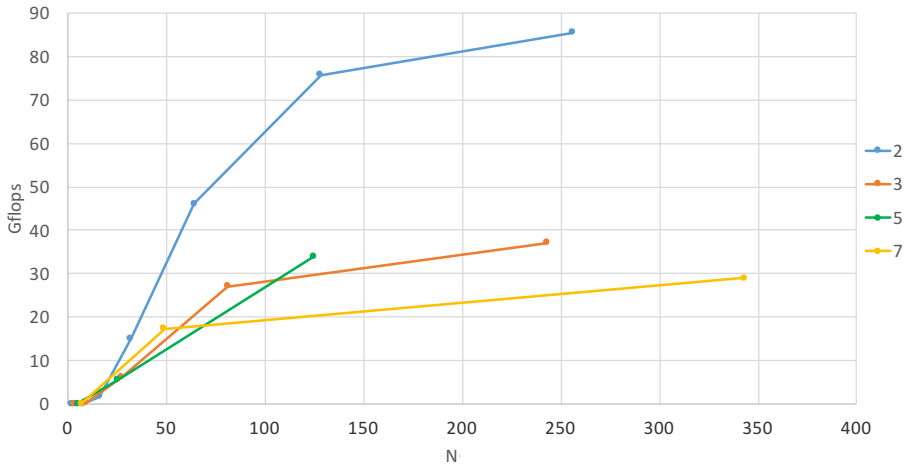


Figure 5.2: 3D cuFFT speed for different sizes and powers (Nvidia GTX 780 with 166 Gflops of theoretical peak power, double precision, real valued transforms).

To calculate the speed of FFT routines (in flops, floating-point operations per second) it's necessary to know the number of operations performed in a certain amount of time. The time has to be the *wall clock time*, that is the effective time occupied by the computation; one way to measure it is by using the function:

```
gettimeofday(struct timeval *tv, struct timezone *tz);
```

on Unix operative systems, available in the `sys/time.h` library. On Windows systems, a slightly less convenient way of measuring the time between two CUDA events can be implemented using functions like `cudaEventCreate`, `cudaEventRecord`, `cudaEventSynchronize` and `cudaEventElapsedTime`.

Regarding the number of operations performed by a FFT, the complexity of Cooley-Tuckey's algorithms varies with the prime factor of the dimension's size. The operations executed by a FFT can be calculated [KAU98] as:

$5/2 * n * log_2(n)$	for radix-2 algorithms
$9.33/2 * n * log_5(n)$	for radix-3 algorithms
$13.6/2 * n * log_5(n)$	for radix-5 algorithms

where n is the size of the transform (n^3 in 3D); the division by 2 is present because real valued transforms execute half the operations compared to complex valued transforms.

These results can just be considered as estimates, since in the majority of FFT libraries the number of operations is inferior; the *Gflops* are not an absolute indicator of the processor's efficiency, but rather a convenient scale factor to compare performances.

Multi-dimensional array layout

The cuFFT library (like the FFTW library) is programmed to handle multi-dimensional arrays stored in a *row-major* order in memory. Since computer memory in inherently linear, mapping multi-dimensional data on in can be done in several ways; the two most popular memory layout are *row-major* (used in C, C++, Mathematica, Python) and *column-major* (used in Fortran, Matlab, Octave).

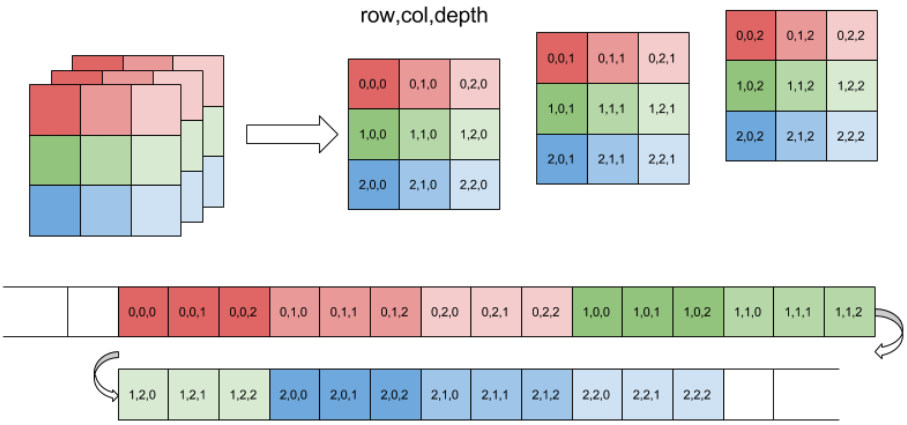


Figure 5.3: Memory layout of a 3D array with $n_x = n_y = n_z = 3$ in row-major order. [Ben15]

When working with 2D arrays the layout are simple to describe: row-major puts the first row of the matrix in contiguous memory, then the second row after it and so on; column-major instead puts the first column in contiguous memory, then the second column after it and so on.

In three dimensions it's a bit more complicated. Fig. 5.3 shows the mapping from 3D arrays to linear memory in row-major order. It's the last dimension (depth) that changes the fastest and the first dimension (row) the changes the slowest. The offset of a certain element of the array, with respect to the memory address of the first element, is computed with:

$$offset(i_x, i_y, i_z) = i_z + n_z * (i_y + n_y * i_x), \tag{5.1}$$

where i_x,i_y and i_z are the indices that define the element's position in the 3D array. In column-major order however, the slowest changing index is depth while the fastest changing is row.

In two dimensions the offset is simply reduced to:

$$offset(i_x, i_y) = i_y + n_y * i_x, \tag{5.2}$$

The practical rule is to traverse the data in the order it was laid out, for the purpose of achieving aligned memory access and exploit spatial locality (Section 2.4.1).

Padding

The only currently supported data layout for in-place transforms in cuFFT is *padded*: the output data begin at the same memory address as the input data, therefore input data for real-to-complex transforms and output data for complex-to-real transforms must be padded because the complex array is larger than the real array, and the two share the memory location. The last dimension (z in 3D, y in 2D) of the real-data array must physically contain $2 * (n_z + 1)$ `double` values, enough to hold $n_z + 1$ `cuDoubleComplex` values. The illustration in Fig. 5.4 describes this memory configuration.

When accessing the data in the physical domain for initialisation or printing, the programmer has to be careful to avoid memory locations that has been padded and are not part of the actual real array. This means that in the last dimension there are $2 * N_z$ elements of the real array that must not be accessed: they are the last 2 elements in the last dimension (depth in 3D, column in 2D). To access only relevant elements, the program contains the `N_PAD` variable that stores the padded dimension: $N_{pad} = 2 * (n_z/2 + 1)$ in 3D and $N_{pad} = 2 * (n_y/2 + 1)$ in 2D. The offsets in Eqs. (5.1) and (5.2) then change to:

$$offset_{pad}(i_x, i_y, i_z) = i_z + n_{pad} * (i_y + n_y * i_x) \qquad \text{in 3D arrays}$$
$$offset_{pad}(i_x, i_y) = i_y + n_{pad} * i_x \qquad \text{in 2D arrays}$$

5.2.5 Memory allocation

The memory configuration of the GPU determines the ability to allocate data. Since all the data must be contained in the global memory (VRAM, video RAM), it's important not to occupy more memory than what is available on the device or the program won't run. What limits the program is the number of modes of the expanded array, needed in order to avoid aliasing.

The fast Fourier transforms of the velocity and non-linear data are computed in-place to better utilise the scarce available memory. Performance implications are discussed in Section 6.5.

The declarations of the most important variables that requires allocation is:

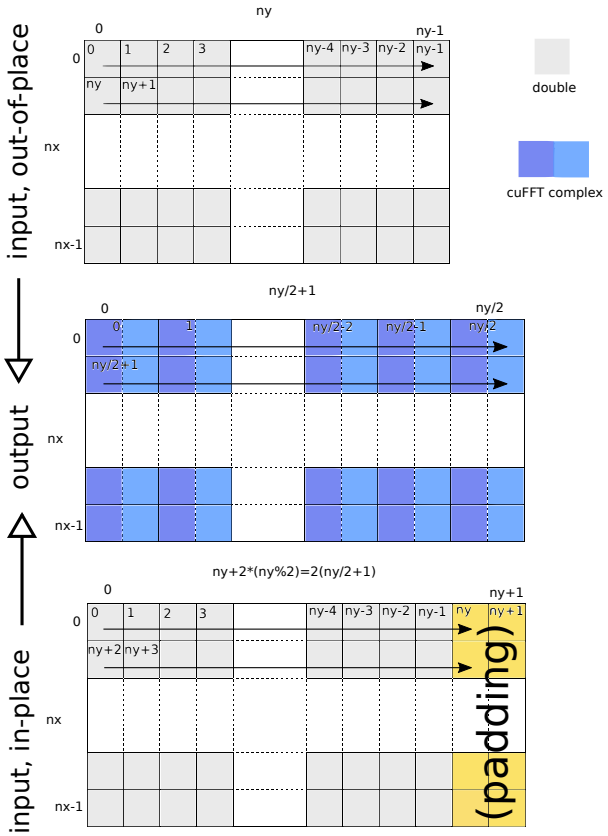


Figure 5.4: Input and output array for the in-place and out-of-place transforms in 2D for the FFTW library, which is equal to cuFFT (the arrows indicate consecutive memory locations)[FJ14a]

```
struct velocityReal *h_VR, *d_VR, *h_VR_sol;
struct velocityComp *h_V, *d_V, *h_V_sol;
struct nonLinearReal *h_NLR, *d_NLR;
struct nonLinearComp *h_NL, *d_NL;
struct rhsComp *h_OLD, *d_OLD;
double *h_EcinP_vect, *d_EcinP_vect;
```

To easily distinguish device variables from host variables, it's a good practice to put `d_` in front of device variables' names, and `h_` in front of host variables' names.

To allocate memory for the GPU, the CUDA toolkit provides a function with a syntax similar to the ANSI C `malloc`:

```
cudaError_t cudaMalloc(void **device_ptr, size_t size);
```

that allocates `size` bytes of linear memory and returns, in `*device_ptr`, a pointer to the allocated device memory. It's common practice to cast the pointer to the variable's type directly in this function, for example:

```
cudaMalloc((struct velocity **)&d_V, sizeof(*d_V));
```

Device variables needs to be deallocated using:

```
cudaError_t cudaFree(void **device_ptr);
```

Due to the fact that the FFTs of velocity and non-linear data are executed in-place, there are effectively only 4 variables that needs allocation on GPU memory: `d_V`, `d_NL`, `d_OLD` and `d_EcinP_vect`. The physical domain velocity structure and the physical non-linear structure are obtained as a re-cast of the respective complex variables:

```
d_VR = (struct velocityReal *) d_V;
d_NLR = (struct nonLinearReal *) d_NL;
```

and don't need to be de-allocated. The host-side respective variables are allocated in the classical ANSI C way; the two variables `h_V_sol` and `h_VR_sol` are allocated independently, so it's possible to store the physical and complex solution at the same time.

As detailed before in Section 5.2.3, the velocity and non-linear terms data structures occupy respectively 96 and 48 bytes for each element, but each field of the non-linear terms structure has much more elements that the fields of the velocity structure, and it will be the limiting factor for the dimension of the problem that can be resolved on the GPU.

N	N_{dir}	N_{exp}	velocity complex	non-linear complex	rhs complex	kinetic energy
20	41	64	1.5 MB	12.4 MB	1.5 MB	0.3 MB
42	85	128	13.9 MB	97.5 MB	13.9 MB	2.3 MB
84	169	256	113.8 MB	774 MB	113.8 MB	14.3 MB
170	341	512	904.9 MB	6168 MB	904.9 MB	150.8 MB

Table 5.2: Memory occupied by user declared data

In Table 5.2 there are some examples of memory occupancy from the data declared by the program. By summing the data presented in this table with the ones in Table 5.1, it's possible to make an estimate of the memory occupied by the program, and thus understand the size of the problem that can be resolved with the hardware available to the user. For example, a GPU with 1GB of VRAM can handle a physical velocity domain of 84^3 with a de-aliasing matrix of 128^3; a physical velocity domain of 170^3 with a de-aliasing matrix of 256^3 requires 3GB of VRAM in out-of-place mode, but a GPU with 2GB of VRAM can handle this domain size in in-place mode.

The program however contains other variables that need to be stored in global memory other than data structures and execution plans. A precise measurement of the memory utilisation is obtained with the CUDA function:

```
cudaError_t cudaMemGetInfo(size_t *free_mem, size_t *total_mem);
```

5.2.6 Initial conditions

As explained before in Section 4.4.5, there are two possibilities for the initialisation of the program: the Taylor-Green vortex and the divergence-free condition.

The Taylor-Green vortex is a classical method to validate DNS solvers because of the existence of an analytical solution. The main file contains the function `void TaylorGreen_initial_conditions()` that includes a nested loop to create the physical initial conditions in the real velocity structure fields `VR->u`, `VR->v` and `VR->w`; at the same time, the velocity array elements are *pre-normalised*, by dividing for the transform size `DOM_SIZE_R`, for the following FFTs.

Before executing the Fourier transforms each component need to be treated to prevent an undesired feature of the cuFFT library: this operation is called centring, more on this in Section 5.7.

The data is then copied to GPU memory and three in-place FFTs transform the physical data to the wavenumbers domain variables `V->u`, `V->v` and `V->w` used by the program.

The divergence-free initial condition is simpler to implement because it's already given in the wavenumbers domain. The main file contains the function:

```
void LCP_inintial_conditions(unsigned int k2[DOM_SIZE_C],
    double k_mod[DOM_SIZE_C]);
```

that requires κ^2 and $|\kappa|$ as inputs (previously computed in the main function), and includes a nested loop that fills the fields `V->u`, `V->v` and `V->w` as detailed in Eqs. (4.16) and (4.17). The complex data is then copied to GPU memory.

The function that handles the copying of data to and from GPU memory is:

```
cudaError_t cudaMemcpy(void *dest_ptr, const void *src_ptr,
    size_t size, enum cudaMemcpyKind kind);
```

It copies `size` bytes from the memory area pointed to by `src_ptr` to the memory area pointed to by `dest_ptr`; `kind` specifies the transfer direction (primarily `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`).

5.3 Grids configuration

CUDA programs are designed to work on a large number of graphical processing units, everything Nvidia released in the last 5 year; only the first CUDA-capable GPUs are not supported anymore, but they don't even support double precision computation.

To adapt a program to a target device, it's possible to modify the *execution configuration* of the kernels in order to harvest the GPU resources in the best way. The execution configuration specifies how threads are scheduled to run on the GPU.

By specifying the grid and block dimensions, the programmer effectively configures the total number of thread for a kernel and their layout. However, there are limitations on grid and block dimensions: the maximum sizes at each level of the thread hierarchy is device dependent, and depends on the compute capability of the device.

In order to decide the best execution configuration, a useful tool is the CUDA Occupancy Calculator, a spreadsheet created by Nvidia to calculate kernel's occupancy (Section 2.6). The input needed are: compute capability of the GPU, threads per block, registers per thread and shared memory per block. These last two informations can be obtained by compiling with the flag `--ptxas-options=-v`.

To enhance occupancy, it's possible to resize the block configuration or re-adjust resource usage to permit more simultaneously active warps and improve utilization of compute resources. Manipulating thread blocks to either extreme can restrict resource utilization:

- small thread blocks: too few threads per block leads to hardware limits on the number of warps per SM to be reached before all resources are fully utilized;

- large thread blocks: too many threads per block leads to fewer per-SM hardware resources available to each thread.

The most important guideline to follow is to keep the number of threads per block a multiple of warp size (32 threads); moreover, the number of blocks should be much greater than the number of SMs to expose sufficient parallelism to the device.

In practical terms however it's best to experiment with different configurations, since the Nvidia compiler perform a great number of optimisation and it's impossible to judge in advance even the register usage of a kernel, much less the best execution configuration.

In figure Fig. 5.5 it's possible to see a plot of the execution time for the `copyV2NLcomplex_and_kineticEnergy()` kernel and how it varies with block size. By keeping a low number of threads per block (64) more blocks are present in the grid therefore more parallelism is exposed.

The Occupancy Calculator shows that with 64 threads per block the occupancy is just 50%, while for 1024 thread is 100%, however the configuration with 1024 threads is 20% slower than the one with 64 threads. Outside the extremes the situation becomes difficult to predict, but it's a balance between the higher number of blocks per SM towards the left of the

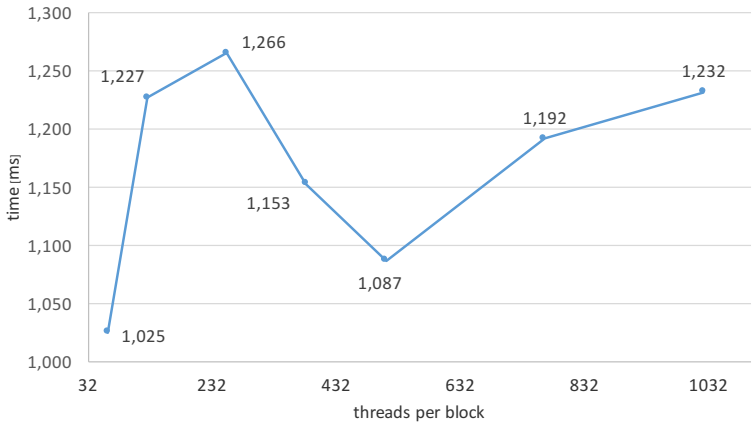


Figure 5.5: Velocity expansion and kinetic energy kernel execution time on an Nvidia GTX780; 169^3 modes, 32 registers.

thread axis (more parallelism), and a higher number of warps per block to the right (more threads to better hide latency).

After several experiments, the best execution configurations has been found for the kernels. Except for the non-linear products kernel, all configurations are 1D grids of 1D blocks; it's been decided this because while the number of elements to handle is big, it's not so big that it doesn't fit inside a grid or block direction limitation ([Nvi15a]): blocks have maximum dimensions of $(1024 * 102 + x64)$ but can contain no more than 1024 threads, grids have maximum dimensions of $(65535 * 65535 * 65535)$ for devices with compute capabilities $2.x$ and $((2^{31} - 1) * 65535 * 65535)$ for compute capabilities 3.0 or greater.

For the non-linear products kernel it's been decided to employ a 2D grid of 1D blocks, due to the fact that for $N = 85$ there are 16777216 elements in each field array, and a 1D grid would require blocks of at least 256 threads. To leave the possibility to employ smaller blocks, a 2D grid is configured for this kernel.

The best thread block sizes are included as macros in the header file. The kernels' grid dimensions are obtained from the block size with an integer division involving the number of elements that the kernel needs to handle, as can be seen in these examples:

```
// 1D blocks
dim3 block_red(TPB_RED, 1, 1);
dim3 block_prod(TPB_PROD, 1, 1);

// 1D grid
dim3 grid_red((DOM_SIZE_C+block_red.x-1)/block_red.x, 1, 1);
// 2D grid
dim3 grid_prod((EXP_DOM_SIZE_R/block_prod.x+block_prod.x-1)
```

```
/block_prod.x, block_prod.x, 1);
```

5.4 Indices

To avoid creating arrays to contain the indices of Fourier modes $\{l,m,n\}$ that would have to be read by the GPU, several macros are used to obtain these values from the thread index variable that is fundamental for every kernel. In the GPU, memory transactions are expensive while floating-point operations are cheap so this is the fastest way to have the three indices always available. For example, the performance loss of the velocity computation kernel that relies on indices arrays it's been appraised to be about 9%.

The key variable to calculate $\{l,m,n\}$ is the global thread index `tIdx`. For kernels that operate on a 1D grid on 1D block the index of a the single thread is computed with:

```
unsigned int tIdx = threadIdx.x + blockIdx.x * blockDim.x;
```

For the non-linear product kernel that operates on a 2D grid of 1D blocks, the index is obtained with:

```
unsigned int blockId = blockIdx.y * gridDim.x + blockIdx.x;
unsigned int tIdx = blockId * blockDim.x + threadIdx.x;
```

The index variable's type is big enough to handle the maximum number of threads in common problem sizes: in a 64-bit system, the `unsigned int` variable has a size of 4 bytes, which translates to the range of values $(0, 2^{32}) = (0, 4294967296)$, that is big enough to handle a de-aliased non-linear terms matrix of 1024^3; it's unnecessary to employ a `unsigned long int` variable which would only waste register memory on the GPU. The built-in `blockIdx` and `threadIdx` variables are directly provided by the CUDA toolkit, as explained in Section 2.4.2.

First, the three macros `L_IDX_from0`, `M_IDX_from0` and `N_IDX_from0` calculate $\{l,m,n\}$, strictly in this order, in the range $(0, 2N)$ using the dimensions of the **complex** velocity matrix and the knowledge of the row-major layout of the memory (Section 5.2.4):

$$\begin{cases} l_{f0} = \dfrac{tIdx}{Ncols * Ndeps} \\ m_{f0} = \dfrac{tIdx - l_{f0} * Ncols * Ndeps}{Ndeps} \\ n_{f0} = tIdx - Ndeps * (m_{f0} + l_{f0} * Ncols) \end{cases}$$

What makes possible to obtain the indices is the fact that these equations are actually divisions between integer variables, thus the output is only the *integer* part of the result.

The equations however utilise these indices in the range $(-N, N)$, so it's just a matter of subtracting N from l_{f0}, m_{f0} and n_{f0} to obtain the macros `L_IDX`, `M_IDX` and `N_IDX`. Because the program also handles 2D problems, it's important to differentiate the n index calculation for this case where it would be $n = 0$.

Another set of macros is used to map the velocity array elements to the corresponding expanded non-linear terms array elements, needed for de-aliasing purposes.

As is discussed in Section 5.7, the cuFFT library's transforms put the axis origin in the zero-th element of the array per convention.

Reordering the non-linear terms spectrum would be a significant waste of time, instead is much faster to follow the library convention and find a way to deal with this reordering.

It's important to create an *interface* for the program to work with data in this ordering; this is done with three new macros:

```
#define L_EXP_IDX ( (L_IDX<0) ? (N_EXP+L_IDX) : (L_IDX) )
#define M_EXP_IDX ( (Nz==1)?M_IDX:((M_IDX<0)?(N_EXP+M_IDX):(M_IDX)) )
#define N_EXP_IDX ( (Nz==1) ? 0 : N_IDX )
```

The expanded index `EXP_IDX` is then:

```
N_EXP_IDX + N_DEPs_EXP_C*(M_EXP_IDX + L_EXP_IDX*N_COLs_EXP_C)
```

This global index is used in both kernels that compute the velocity field in order to pick the correct de-aliased non-linear products among the full array and discard the spurious terms.

5.5 Kernels

The `cuNS_kernels.cu` file contains the user programmed kernel functions that are executed on the GPU; it also calls the `defines.h` header file, so it has access at the same macros as the main file.

In the following section, the kernels are discussed.

Vector expansion and kinetic energy contributions

To copy the velocity array elements in the non-linear array as illustrated in Fig. 4.6, the following kernel is called:

```
__global__ void copyV2NLcomplex_and_kineticEnergy(struct velocityComp
    *vel, struct nonLinearComp *nonL, double *EcinPvect, double k_f);
```

The `__global__` keyword identifies a function that is called by the host but executes on the device; another option for a function is the `__device__` keyword, for functions that executes on the device and are called by it.

The GPU threads map the elements of the velocity terms, so it's fundamental to insert a boundary condition check to avoid executing illegal global memory access:

```
if (tIdx >= DOM_SIZE_C) return;
```

To reduce the negative effects of warp divergence (Section 2.6), it's better to insert simple instructions inside a conditional clause and therefore, in this case, check which threads are *outside* the desired domain (very few threads and a simple instructions in the `if`) compared to check the threads which are *inside* the boundary (many more threads and complex instructions inside the `if` in this way).

Conceptually, the calculation of kinetic energy should be a separate step from the velocity expansion. However, since it requires the same inputs (elements of the velocity structure), it's convenient to include such calculation in the same kernel.

After making sure that the case with $l = m = n = 0$ is avoided, the single contribution to the kinetic energy of the flow is stored in the array `EcinPvect` only if the wavenumber of the mode belongs to the large scale structures where the forcing occurs, that is $|\boldsymbol{\kappa}| \leq \kappa_f$. This array needs to be summed to a single kinetic energy values in order to use it as forcing term.

Kinetic energy array reduction

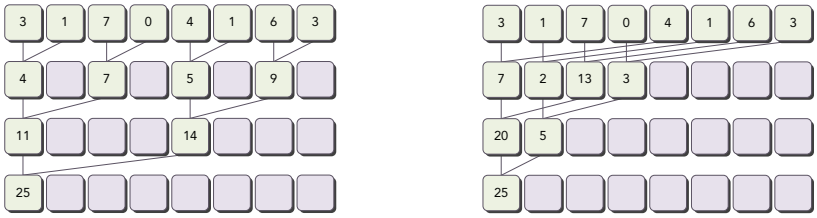
Summing the elements of an array (an operation called *reduction*) is a trivial task on a CPU with serial code; a parallel reduction however requires more effort than a simple loop to obtain good efficiency. It's one of the most common parallel patterns, and a key operation in many parallel algorithms.

A common way to obtain parallel reduction is a pairwise implementation:

1. partition the input array in small chunks, each containing only a pair of elements
2. have a thread sum the two elements to produce a partial result
3. store the partial result *in-place* in the original input vector
4. use the new values as input for the next iteration of the reduction

Depending on where output elements are stored in-place for each every iteration, pairwise parallel sum implementations can be classified in two types, also illustrated in Fig. 5.6:

- *neighboured* pair: elements are paired with their immediate neighbour;
- *interleaved* pair: paired elements are separated by a given stride;



(a) Neighboured pair **(b)** Interleaved pair

Figure 5.6: Pairwise parallel sum implementations. [CGM14]

Generally, the interleaved implementation is faster thanks to more efficient global memory load and store patterns (aligned load and store operations).

The method employed in this work is a hybrid parallel-serial reduction, as showed in Fig. 5.7. This allows to exploit the massive parallelism of the GPU to sum the array elements until only one element per block remains. A data transfer is then issued to copy `gridDim.x` elements from GPU memory to CPU memory and then execute a serial reduction on the CPU, on a much lower number of elements compared to a reduction on the entire kinetic energy contributions array.

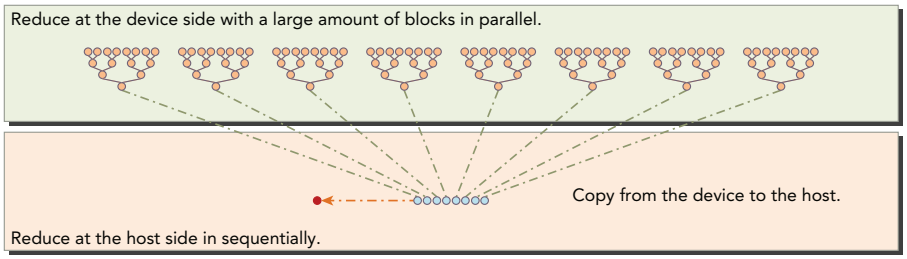


Figure 5.7: Hybrid GPU-CPU array reduction. [CGM14]

A way to improve performance in this case is *loop unrolling*, a technique that attempts to optimise loop execution by reducing the frequency of branches and loop maintenance instructions. In loop unrolling, the body of a loop is written multiple times, thus every loop has a reduced number of iterations. The number of copies made of the loop body is called *unrolling factor*; the number of iterations is divided by the loop unrolling factor. The performance improvements come from reduced instruction overheads and more independent instructions scheduled. As a result, more concurrent operations are added to the pipeline, leading to higher saturation of instruction and memory bandwidth; this provide the warp scheduler with more eligible warps that can help hide instruction or memory latency.

For the reduction kernel, the basic idea is to make each thread handle

two portion of the data (data block):

```
if (idx+blockDim.x < n) g_idata[idx] += g_idata[idx+blockDim.x];
__syncthreads();
```

where the block-local barrier `__syncthread` is required to be sure that all the threads in a block are at the same level of execution; `n` is the input data size, and the global index requires and adjustment as:

```
unsigned int idx = blockIdx.x * blockDim.x * 2 + threadIdx.x;
```

because only half as many thread blocks needs to be processed; this also implies that less parallelism is exposed to the device.

Because each thread block now handles two data blocks, the execution configuration of the kernel must be adjusted by reducing the grid size by the unrolling factor (by half in this case).

The unrolling factor can be higher than two, however the improvements slow down as the unrolling factor increases; in this work, a kernel with un unrolling factor of four is implemented.

It's important to handle the case where there are 32 or fewer threads left: a single warp. Because warp execution is SIMT (Section 2.2), there is implicit intra-warp synchronisation after each instruction. The last 6 iterations of the reduction loop can therefore be unrolled in the following way:

```
if (threadIdx.x < 32){
    volatile double *vmem = idata;
    vmem[threadIdx.x] += vmem[threadIdx.x + 32];
    vmem[threadIdx.x] += vmem[threadIdx.x + 16];
    vmem[threadIdx.x] += vmem[threadIdx.x + 8];
    vmem[threadIdx.x] += vmem[threadIdx.x + 4];
    vmem[threadIdx.x] += vmem[threadIdx.x + 2];
    vmem[threadIdx.x] += vmem[threadIdx.x + 1];
}
```

This warp unrolling avoids executing loop control and thread synchronisation logic.

The declaration of the `vmem` variable with the `volatile` qualifier is essential, because it tells the compiler to store `vmem[threadIdx.x]` back to global memory with every assignment (and avoid optimisations like placing it in a register), because its value can be changed or used at any time by other threads.

Moreover, if the number of iterations in a loop is know at compile-time, it's possible to completely unroll it. Because the maximum number of threads per block is limited to 1024, and the loop iteration count in the reduction kernel is based on the thread block dimension, this information is known and the complete unrolling starts with:

```
if (blockDim.x>=1024 && threadIdx.x<512)
    g_idata[threadIdx.x] += g_idata[threadIdx.x+512];
__syncthreads();
```

and continues with similar instructions by halving 1024 an 512 until the local thread index is greater than 32, then the warp unroll finishes the work as explained before.

Another method of improving performance in the reduction kernel is by using shared memory. One of the main reason to use it is to cache data on-chip, therefore reducing the number of global memory accesses in the kernel. In this case the kernel would be nearly identical to one based on global memory, however instead of using a subset of the global memory input array to perform an in-place reduction, a shared memory array is declared with the proper keyword:

```
__shared__ double smem[DIM];
```

this array **must** have the same dimensions as each thread block. Each thread block initialises the shared memory with the portion of global memory it handles:

```
// convert global data pointer to the local pointer of this block
double *idata = g_idata + blockIdx.x * blockDim.x;
// set to shared_mem by each threads
smem[threadIdx.x] = idata[threadIdx.x];
```

Then, the in-place reduction is performed using shared memory (`smem`) rather than global memory (`idata`).

Two reduction methods are included in the kernels' file:

```
__global__ void reduction_shared_mem(double *global_in_data,
    double *global_out_data, unsigned int n);

__global__ void reduction_shared_mem_unrolled4(double *global_in_data,
    double *global_out_data, unsigned int n);
```

The inputs for both kernel functions are: pointer to the array to be reduces, pointer to the reduced array and number of elements to sum. Both kernels have the boundary condition that is the number of element to reduce.

The choice between the two implementations is automatic: if the grid size is a multiple of four, the unrolled 4-times version is chosen. It's approximately twice as fast compared the non-unrolled version. The main differences between the two are these lines:

```
// global index, 4 blocks of input data processed at a time
unsigned int idx = blockIdx.x * blockDim.x * 4 + threadIdx.x;

// unrolling 4 blocks
double temp_sum = 0;
if (idx < n){
    double a1 = 0, a2 = 0, a3 = 0, a4 = 0;
    a1 = global_in_data[idx];
    if (idx+blockDim.x < n) a2 = global_in_data[idx+blockDim.x];
    if (idx+2*blockDim.x < n) a3 = global_in_data[idx+2*blockDim.x];
```

```
        if (idx+3*blockDim.x < n) a4 = global_in_data[idx+3*blockDim.x];
        temp_sum = a1 + a2 + a3 + a4;
    }
    shared_mem[threadIdx.x] = temp_sum;
    __syncthreads();
```

Note the thread index that handles 4 blocks at the same time.

Non-linear products

After the velocity array expansion, the non-linear terms lie in the wavenumbers domain. Three (two in 2D) double precision Inverse Fourier Transforms (IFTs) are performed to bring them in the physical domain, placing the results in the `d_NLR` structure.

It's then time for the cross-products of the non-linear terms: the computation is in the kernel:

```
__global__ void NLR_products(struct nonlinearReal *nonLR);
```

This kernel simply executes the six cross-products (three in 2D) and at the same time applies a pre-normalisation by dividing for the expanded transform size `EXP_DOM_SIZE_R`.

The boundary condition for this kernel is the size of the physical expanded domain size `EXP_DOM_SIZE_R`.

After the kernel, six (three in 2D) FFTs are performed to bring the non-linear products back to the wavenumbers domain: this effectively is a faster way to compute a convolution for a large number of elements.

Velocity field computation

As detailed in Section 4.4.4, there are two ways to compute the velocity field through time discretisation: exact or semi-implicit treatment; they are included in two separate kernel functions that can be selected in the main file, of which prototypes are:

```
__global__ void exact_treatment(struct velocityComp *vel, struct
    nonLinearComp *nonL, struct rhsComp *old, double EcinP, double k_f);

__global__ void approx_treatment(struct velocityComp *vel, struct
    nonLinearComp *nonL, struct rhsComp *old, double EcinP, double k_f);
```

The third input field is the structure needed to contain the non-linear terms of the previous iteration for use in multi-step time discretisation methods. The fourth input is the actual kinetic energy obtained from the previous array reduction; the last input field contains the large scale structures's wavenumber, which characterise the length scales over which the forcing is exerted.

The boundary condition for these kernels is the wavenumber domain size `DOM_SIZE_C`.

Both kernels share a common part to calculate the non-linear terms (Eq. (4.12)) and then extract the perpendicular component with respect to the wavenumber vector $\boldsymbol{\kappa}$ using the *projection tensor* operator (Eq. (4.10)). It would seem better to gather these calculations in a separate kernel, but that would add latency from the kernel launch and overhead in general; performance-wise, it wouldn't provide faster execution, thus it was decided to replicate the same commands in both kernels.

Particular attention has to be given to these computations. If these lines of code had to be written in the ANSI C language by using the `complex.h` library (designed to work with complex numbers), the commands would be similar to:

```
uu = I*l* uu + I*m*uv + I*n*uw;
```

using the imaginary unit `I` built in the library, both real and imaginary fields are assigned correctly in one command. However, the CUDA toolkit used in this work (version 7.0, released in March 2015), with its library `cuComplex.h`, doesn't support this syntax. Instead, it implements complex numbers as a C vector type with real and imaginary fields, so the previous commands has to be separated in two commands, dividing the real and complex part of the result:

```
uu.x = -l*uu.y - m*uv.y - n*uw.y;
uu.y =  l*uu.x + m*uv.x + n*uw.x;
```

This however is not correct: by assigning a new value to the real component of uu in the first line, the second line uses this new value to calculate the imaginary part instead of the original value of `uu.x` that comes from the FFT executed before. This can be solved by declaring three new variables `uu_temp` , `vv_temp`, `ww_temp` of type `cuDoubleComplex` inside the kernels, so that the proper commands are similar to the following ones:

```
uu_temp.x = -l*uu.y - m*uv.y - n*uw.y;
uu_temp.y =  l*uu.x + m*uv.x + n*uw.x;

vv_temp.x = -l*uv.y - m*vv.y - n*vw.y;
vv_temp.y =  l*uv.x + m*vv.x + n*vw.x;

ww_temp.x = -l*uw.y - m*vw.y - n*ww.y;
ww_temp.y =  l*uw.x + m*vw.x + n*ww.x;

// Perpendicular component
uu.x = -(1.0/k2) * ( (k2-l*l)*uu_temp.x - l*m*vv_temp.x - l*n*ww_temp.x );
uu.y = -(1.0/k2) * ( (k2-l*l)*uu_temp.y - l*m*vv_temp.y - l*n*ww_temp.y );

vv.x = -(1.0/k2) * ( -l*m*uu_temp.x + (k2-m*m)*vv_temp.x - m*n*ww_temp.x );
vv.y = -(1.0/k2) * ( -l*m*uu_temp.y + (k2-m*m)*vv_temp.y - m*n*ww_temp.y );

ww.x = -(1.0/k2) * ( -l*n*uu_temp.x - m*n*vv_temp.x + (k2-n*n)*ww_temp.x );
ww.y = -(1.0/k2) * ( -l*n*uu_temp.y - m*n*vv_temp.y + (k2-n*n)*ww_temp.y );
```

After these identical lines for both kernels, they start to differ. The first kernel solves the exact equations (Eq. (4.14)), while the second contains the semi-implicit approximation (Eq. (4.15)).

At the end, both kernels store in the structure `old` the variables that multiply the constant b in Eqs. (4.14) and (4.15).

The constants a, b, c and d that decide which discretisation method (Adam-Bashforth 2, Crank-Nicolson, explicit Euler) is used in these kernels are included in the header file.

5.6 Asynchronous operations

Every kernel is *asynchronous* with respect to the host, meaning that after the launch it goes on a queue, waiting for previous kernels to end, while the CPU can do any kind of work unrelated to GPU computations. While it is possible to run multiple kernels at the same time, all the GPU resources (like registers and shared memory) would have to be split between the concurrent kernels. Moreover, in this program only the reduction and the non-linear products kernels don't depend on each other, and the latter requires a big grid since it has many more elements than the former, so the resource partitioning would be counter-efficient.

It makes more sense in this case to overlap memory transfers and kernel executions, thanks to the ability to create multiple *streams*. A CUDA stream is defined as a sequence of asynchronous CUDA operations that execute on a device in the order issued by the host code. Operations in different streams have no restrictions on execution order: multiple streams that launch multiple kernels effectively implements *grid level concurrency*.

In many cases, more times is spent executing the kernel than transferring data, so a proper use of multiple streams can hide CPU-GPU communication latency. By dispatching kernel executions and data transfer into separate streams, these operations can be overlapped, reducing the program's execution time.

The functions in the CUDA API can be classified as either synchronous or asynchronous. Functions with *synchronous behaviour* block the host until they complete, while functions with *asynchronous behaviour* return control to the host immediately after being called.

All CUDA operations either explicitly or implicitly run in a stream. There are two types of streams:

- implicitly declared stream (NULL stream): the default stream that kernel launches and data transfers use;

- explicitly declared stream (non-NULL stream): needs to be created and managed, required to overlap operations;

Asynchronous, stream-based kernel launches and data transfers enable four types of *course-gran concurrency*:

- overlapped host computation and device computation
- overlapped host computation and host-device data transfer
- overlapped host-device data transfer and device computation
- concurrent device computation

Kernel launches are always asynchronous. Data transfers can be issued asynchronously by specifying the CUDA stream to place them in with the function:

```
cudaError_t cudaMemcpyAsync(void *dst, const void *src, size_t count,
    enum cudaMemcpyKind kind, cudaStream_t stream = 0);
```

the fifth argument is the stream identifier, and by default is set to the NULL (default) stream. The main difference with `cudaMemcpy()` is that this function is asynchronous with respect to the host, so control returns immediately to the host after the call is issued rather than waiting for the copy to be completed. One or more copy engines guarantees the ability to issue one or more data transfers while the GPU is computing data.

In order to associate a copy operation with a non-null stream, it needs to be created with:

```
cudaError_t cudaStreamCreate(cudaStream_t *pStream);
```

where `pStream` is a pointer to a new stream identifier previously declared, of type `cudaStream_t`. The resources of a stream can be released with:

```
cudaError_t cudaStreamDestroy(cudaStream_t pStream);
```

An asynchronous memory copy from or to GPU memory also requires that the corresponding host memory is *pinned* (or page-locked, non-pageable), in order to force its physical location in CPU memory to remain constant through the lifetime of the application. Pinned host memory must be allocated with the following CUDA function:

```
cudaError_t cudaHostAlloc(void **pHost, size_t size, unsigned int flags);
```

which allocates `size` bytes of host memory that is page-locked and accessible to the device; the `flags` parameter enables several options for the allocation. The driver tracks the memory ranges associated with this function and automatically accelerate calls to functions like `cudaMemcpy()`; since the memory can be accessed directly by the device, it can be read or written with higher bandwidth compared to pageable memory obtained with the classical `malloc()`.

Pinned memory must be released with the CUDA function:

```
cudaError_t cudaFreeHost(void *pHost);
```

The CUDA API provides a function that allows to block the host until all operations in a particular stream are completed:

```
cudaError_t cudaStreamSynchronize(cudaStream_t pStream);
```

which is a "lighter" command compared to `cudaDeviceSynchronize()`, which blocks all host operations until the GPU has ended computations.

Non-null streams can be classified in two types: blocking or non-blocking streams. The NULL stream is a implicit stream which synchronises with all other blocking streams in the same CUDA context. Generally, when an operations is issued to the NULL stream, the CUDA context waits on all operations previously issued to all blocking streams before starting that operation.

The CUDA runtime provides a function that allows customisation of a non-NULL stream's behaviour in relation to the NULL stream:

```
cudaError_t cudaStreamCreateWithFlags(cudaStream_t *pStream,
    unsigned int flags);
```

where the `flags` argument specifies the behaviour of the created stream: `cudaStreamDefault` or `cudaStreamNonBlocking`.

In the main program a non-blocking stream is declared and created. After either one of the reduction kernels is executed, a memory copy is issued from device memory to host memory through the user created non-null stream:

```
// After the simple reduction kernel
cudaMemcpyAsync(EcinP_vec_red, d_EcinP_vec_red, sizeof(double)*
    grid_red.x, cudaMemcpyDeviceToHost, stream1);

// After the unrolled 4 times reduction kernel
cudaMemcpyAsync(EcinP_vec_red4U, d_EcinP_vec_red4U, sizeof(double)*
    grid_red4U.x, cudaMemcpyDeviceToHost, stream1);
```

The program then calls the non-linear products kernel which can overlap with these transfers. After the FFTs, there's the CPU part of the hybrid GPU-CPU reduction, which starts with a `cudaStreamSynchronize()` command, to be certain that the GPU partial reduction has ended. A fast serial reduction of length `grid_red.x` or `grid_red4U.x/4` is then executed.

This final CPU work is executed while on the GPU the non-linear products and FFTs are computed in the default stream. For the last kernel, the velocity field computation, it's imperative that the non-linear products and the kinetic energy computations are concluded. For this reason, the `cudaDeviceSynchronize()` is executed before the last kernel call.

An alternative way of obtaining the kinetic energy could be executing a

serial reduction on the CPU while the GPU does other work. It's feasible and even faster than hybrid reduction when the number of elements is low; however, data transfer and serial loop execution takes much more time when N increases.

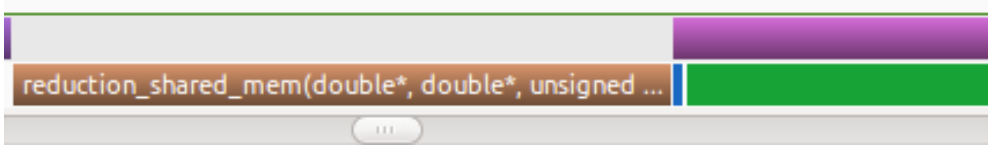


Figure 5.8: Zoom of the Nvidia Visual Profiler generated program timeline.

In Fig. 5.8 there is a crop of a visual representation of the program's GPU work provided by the Nvidia Visual Profiler application (included in the CUDA toolkit), which analyses a program and provides a visual timeline of the duration of all GPU operations. The NULL stream is the first row of the two, the other stream is the second row. The kinetic energy reduction kernel is coloured in red and is on the left side in the non-default *stream1*, after which the results data transfer (in blue) starts, also in stream1. It is evident how short this transfer is compared to transferring the full kinetic energy array (in green) in order to reduce it serially on the gpu. Showing the entire green bar would have rendered other things too small, but for reference it should be noted that it is about three times as long as the reduction kernel; moreover, this big data transfer is shown here for reference, but either this or the reduced array transfer is needed, it is unnecessary to use both.

The asynchronism is clear in this figure: the purple bar represents the non-linear products kernel, and it starts at the same time as the memory transfer but in the default stream, so it runs concurrently to this and to the (potential) full array transfer.

5.7 Other important parts

Non-linear structure reinitialisation

At each iteration the non-linear structure gets modified, but its elements need to be re-initialised at each step since the velocity expansion kernel has to work on empty arrays, otherwise the results from the previous iteration persist and they get transformed by the IFTs, producing a major error. To provide this functionality, the following CUDA function is called at the end of each time step:

```
cudaError_t cudaMemset(void *dev_ptr, int value, size_t count);
```

which fills the first `count` bytes of the memory area pointed to by `dev_ptr` with the constant byte value `value`.

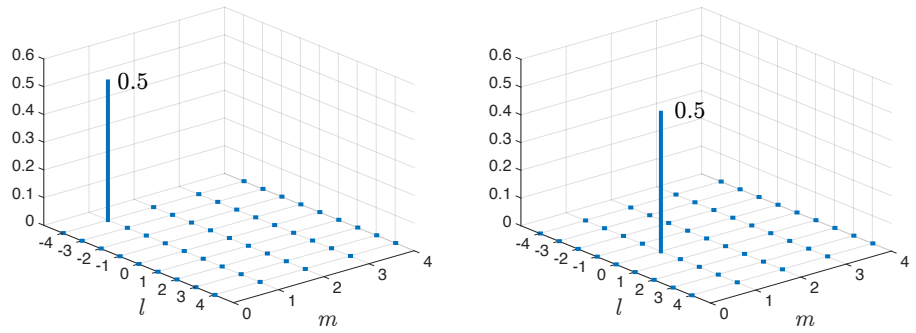
Adapting to the unwanted reordering

The cuFFT library has a particular behaviour regarding the ordering of elements after a transform. By default this library puts the origin in frequency space in the zero-th element of the array, instead that at the centre of the of the output array([FJ14b]). This wouldn't be a problem normally, but in this program the three indices in Fourier space $\{l, m, n\}$ are obtained from the linearised matrix index, so if a peak in the spectrum has the wrong position because of this reordering, its indices will be wrong and the results will not be correct.

The shift of the zero-frequency component to the centre of the array is implemented in the following host function:

```
void reorderV_correctOrder(struct velocityComp *d_vel);
```

that requires the velocity structure, in wavenumber domain, and reorders the spectrum in "human readable" way with the origin at the centre of the 2D or 3D matrix.



(a) without pre-centring **(b)** with pre-centring

Figure 5.9: Spectrum peaks for the modulus of a $cos(y)$ signal after a 2D FFT, without and with the pre-centring treatment, for $N = 4$.

In figure Fig. 5.9 the difference between centred and un-centred output is evident: the peak has indices $(-4, 1)$ in the un-centred matrix, but the centred output peaks are correctly in $(0, 1)$ for the centred one.

Error checking

As it's been seen in previous CUDA function definitions (in Section 5.2.5 for example), they have a particular type: every CUDA call, except kernel launches, returns an error code of an enumerated type `cudaError_t`, indicating either success or details about the failure; this error code can be converted to a human-readable error message with the CUDA runtime function:

```
char* cudaGetErrorString(cudaError_t error)
```

Since many CUDA calls are asynchronous, it may be difficult to identify which command caused an error. For this reason, the header file `defines.h` contains the error-checking macro `CU_ERR_CHECK` designed to wrap all CUDA API calls in order to simplify the error checking process, as in this example:

```
CU_ERR_CHECK( cudaMemcpy(...) );
```

It can be used after a kernel invocation in the following way to check for errors in the kernel:

```
kernel_function<<<grid, block>>>(argument list);
CU_ERR_CHECK( cudaDeviceSynchronize() );
```

the last command blocks the host until the device has completed all preceding requested tasks, and ensures that no errors occurred as part of the last kernel launch. It should be used only for debugging purposes, because this check point is a global barrier.

The cuFFT functions behave somewhat differently regarding error reporting, because they provide another set of codes specific to errors that can happen in FFTs. Every cuFFT function call can be wrapped in the `CU_ERR_CUFFT` macro to collect potential misbehaviours.

5.8 Summary of the time loop

The following pseudo-code shows the order of the critical operations done inside the time loop:

```
// Velocity expansion and kinetic energy array calculation
copyV2NLcomplex_and_kineticEnergy<<<grid, block>>>(...);

// 3 IFTs in 3D or 2 IFTs in 2D
cufftExecZ2D(...);

// Partial kinetic energy array reduction
reduction_shared_mem<<<grid, block, 0, stream1>>>(...);
    or
reduction_shared_mem_unrolled4<<<grid, block, 0, stream1>>>(...);

// Partially reduced kinetic energy array transfer, asynchronous
cudaMemcpyAsync(..., stream1);

// Non-linear products
NLR_products<<<grid, block>>>(...);

// 6 FFTs in 3D or 3 FFTs in 2D
cufftExecZ2D(...);

// Wait for kernels conclusion on stream 1
cudaStreamSynchronize(stream1);

// Finish kinetic energy reduction on the CPU
for (int i=0; ...) EcinP += ...
```

```
// Wait for all opeatiions to conclude
cudaDeviceSynchronize();

// Velocity computation
exact_treatment<<<grid, block>>>(...);
    or
approx_treatment<<<grid, block>>>(...);

// Re-initialisation of the non-linear structure
cudaMemset(d_NL, ...);
```

5.9 After the loop

The loop iterates and the variable `sim_time` is incremented by the value `DT`, contained in the header file, until the final time `Tmax` is reached.

Upon exiting the time loop, the complex velocity structure is copied to the host structure `V_sol`; three (or two in 2D) IFTs are then performed on the GPU to obtain the final physical velocity structure `d_VR`, which is then copied to host memory in the structure `VR_sol`.

Since the complex data is in the reordered form, i.e. the correct and human readable one, these last IFTs would not produce the expected physical results (for the reasons stated before about how the cuFFT library works): a call the the function:

```
void reorderV_FFTOrder(struct velocityComp *d_vel);
```

is needed to prepare the data for the subsequent IFTs by positioning the elements in the order that the library expects them, so that it will give the correct physical result.

The results are then exported to file for following operations like post processing.

If the Taylor-Green initial condition is selected, there's the option to compare the obtained solution to the analytical one by calculating the error: this is useful for validating the code, and is examined in depth in Chapter 6.

The last command of the program, after the de-allocation and plan destructions, the function `cudaDeviceReset()` explicitly destroys and cleans up all resources associated with the current device in the current process, as a way to be sure not to leave unwanted data on the GPU.

6. Validation, results and performance

6.1 Analytical solution

While the Navier-Stokes equations don't have an exact solution, thanks to the work of Taylor and Green [TG37] it's possible to find an analytical solution for a particular condition of the flow.

The Taylor-Green vortex is an exact closed form solution of 2-dimensional, incompressible Navier-Stokes equations. This 2D decaying vortex defined in the square domain $0 < x, y < 2\pi$ is a good benchmark for validating this code; it's also a good problem in which to study vortex dynamics, turbulent transition, turbulent decay and the energy dissipation process.

The Taylor-Green vortex flow has the following initial conditions [Sal11]:

$$u(x, y, 0) = sin(x)cos(y)$$
$$v(x, y, 0) = -cos(x)sin(y)$$

The exact solution is then an exponential decay:

$$u(x, y, 0) = sin(x)cos(y)\ e^{-2\nu t}$$
$$v(x, y, 0) = -cos(x)sin(y)\ e^{-2\nu t}$$

where the speed of decaying is determined by the kinematic viscosity ν.

These equations need to be transformed in the wavenumber domain to be used as initial conditions.

6.2 Validation

The program has been built originally to solve 3D problems and can adapt to 2D ones. However, using six non-linear terms field arrays to solve a 2D problem is a significant waste of memory since only three are needed; in a smaller scale, also using three velocity field arrays when only two are needed is not a smart way of managing resources.

For this reason, a separate version of the code exists to handle only 2D problems in a more efficient manner.

Validating the pseudo-spectral treatment

This code relies heavily on the cuFFT library's accelerated FFTs and IFTs. Moreover, the core of the program is the pseudo-spectral treatment of non-linear products, so it's important that this part of the program is correct.

To verify the correctness of this section it's useful to compare the results of a signal transformation in wavenumbers domain and the multiplication with itself.

As can be seen in Fig. 5.9, the 2D Fourier transform of $u = cos(y)$ is a single peak in the negative half plane $(-l, \ldots, l; -m, \ldots, 0)$; in this case, the y-direction (m index) is the Hermitian direction, so another peak would be present in the other half of the plane.

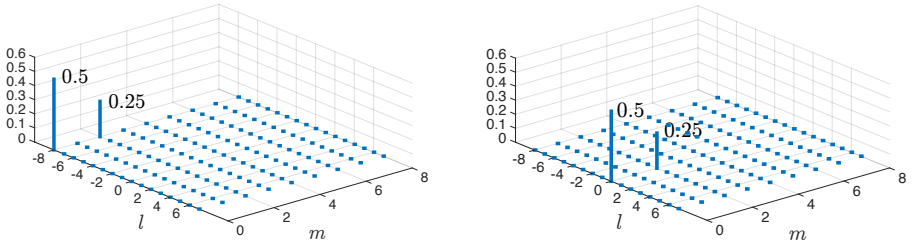
The non-linear product of this signal with itself is:

$$u * u = cos^2(y) = \frac{1 + cos(2y)}{2}$$

after applying trigonometric identities.

The Fourier transform of this signal is composed of one peak for the constant function and one for the harmonic function with a frequency that is double compared to the input signal.

Fig. 6.1a shows the multiplication result in the way that the cuFFT library reorders it. By reordering the data as explained in Section 5.4 it's possible to see in Fig. 6.1b that the two peaks are have been correctly computed (one peak is in the other half plane is not represented).



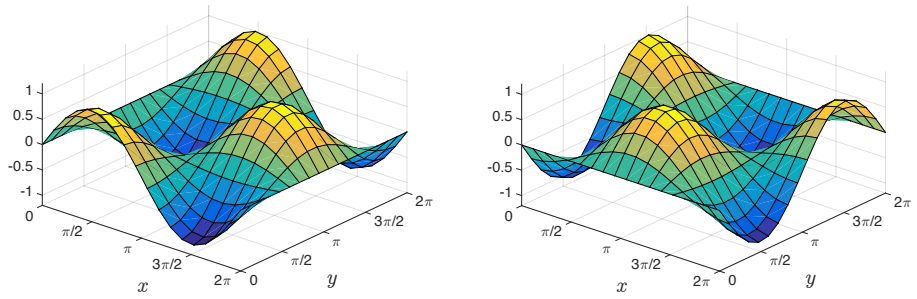
(a) cuFFT ordering **(b)** human readable ordering

Figure 6.1: Spectrum peaks for the modulus of the FFT transform of $cos^2(y)$ after the pseudo-spectral treatment, without and with reordering ($N_{exp} = 8$).

Validation of the 2D version

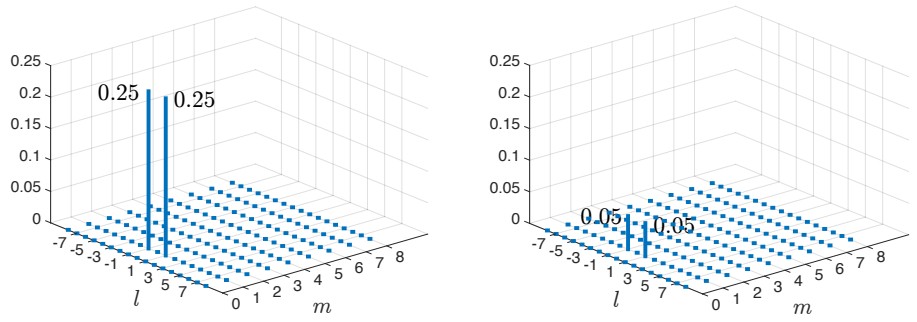
The physical initial conditions for the Taylor-Green vortex are represented in Fig. 6.2 in a domain of size $0 < x, y, < 2\pi$.

This real initial condition is transformed in the wavenumbers domain and then reordered to obtain the complex initial condition represented in Fig. 6.3a.



(a) u component **(b)** v component

Figure 6.2: Taylor-Green vortex flow initial condition in physical domain ($N = 8$).



(a) starting complex data **(b)** complex results after 5s

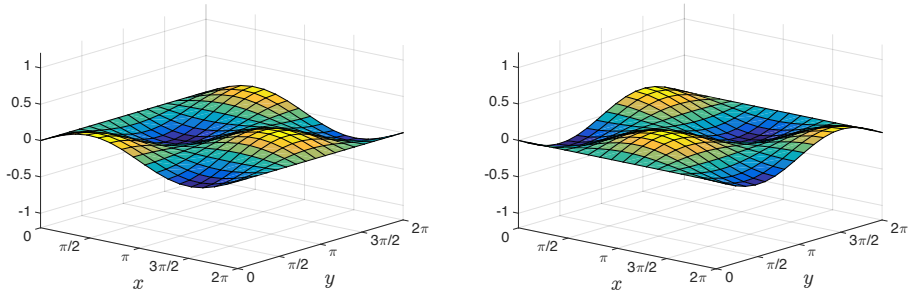
Figure 6.3: Modulus of complex data in wavenumbers domain ($N = 8$).

These initial conditions are characterised by the absence of non-linear terms interactions, so the behaviour of the flow is just an exponential decay governed by the kinematic viscosity ν.

The complex result presents the same two peaks (in each half-plane) but with reduced modulus, as evident in Fig. 6.3b. Also the physical result has the same starting signal but scaled down by the viscosity, as can be seen in Fig. 6.4.

The error is calculated on the physical values of the computed solution compared to the exact solution, but the same calculation can be applied to the Fourier coefficients in the wavenumbers domain; the error in *L2 norm* is calculated as:

$$err_u = \frac{\|u - u_{ex}\|_{L2}}{\|u_{ex}\|_{L2}}$$
$$err_v = \frac{\|v - v_{ex}\|_{L2}}{\|v_{ex}\|_{L2}}$$



(a) u component **(b)** v component

Figure 6.4: Taylor-Green vortex flow computed solution in physical domain after 5s of simulation time (10000 iterations).

where the L2 norm of a generic quantity is defined, in discrete terms, as:

$$\|x\|_{L2} = \left(\sum_{i=-\infty}^{+\infty} x_i^2 \right)^{1/2}$$

The program employs two method to compute the final velocity field: the exact treatment and the semi-implicit approximation (with the Crank-Nicolson method) of viscous and forcing terms; for this reason, the program needs to be validated for both cases.

Table 6.1 gathers the parameters used in the validation tests:

N	Re	Re_t	$\nu[m^2/s]$	Δt	T_{max}	$\hat{f}_\kappa$
8	10	4	1.0000000000e-01	0.0005	10	0
16	19	7.5	5.2631578947e-02	0.0005	10	0
24	28	11.1	3.5714285714e-02	0.0005	10	0
32	38	15.1	2.6315789474e-02	0.0005	10	0
50	60	23.8	1.6666666667e-02	0.0005	10	0
70	80	31.8	1.2500000000e-02	0.0005	10	0
80	108	42.9	9.2592592593e-03	0.0005	10	0
100	201	54.8	7.2463768116e-03	0.0005	10	0
120	168	66.7	5.9523809524e-03	0.0005	10	0
150	234	92.9	4.2735042735e-03	0.0005	10	0

Table 6.1: Parameters for the validation cases in two dimensions.

It's important to notice that the forcing term must be disabled when validating with the Taylor-Green vortex

The results are collected in Table 6.2 and a visual representation of how the evolution changes with the number of Fourier modes employed is in Fig. 6.5.

N_{dir}	N_{exp}	err_u *exact*	err_v *exact*	err_u *approx*	err_v *approx*
17	32	8.00984e-15	7.99527e-15	1.66931e-09	1.66931e-09
33	64	4.85138e-13	4.85128e-13	2.44973e-10	2.44974e-10
49	128	9.63977e-13	9.64028e-13	7.59931e-11	7.59931e-11
65	128	7.58967e-13	7.58977e-13	3.11449e-11	3.11450e-11
101	256	1.52857e-13	1.52860e-13	9.31216e-12	9.31215e-12
141	256	5.48167e-13	5.48263e-13	1.60252e-12	1.60252e-12
161	256	9.60188e-13	9.60194e-13	1.74999e-12	1.74997e-12
201	512	8.30431e-13	8.30417e-13	2.56642e-12	2.56638e-12
241	512	5.02298e-13	5.02558e-13	1.91138e-12	1.91116e-12
301	512	9.48130e-14	9.47911e-14	1.74826e-12	1.74833e-12

Table 6.2: Errors on the u and v component for an increasing number of modes/-points N for two methods of solution; 2D version of the program.

The exact treatment is very accurate even with an extremely low number of modes, and its error remains stable at this level. This is expected, since the equations for this method contains the exponential decaying as $e^{-2\kappa^2 t/Re}$ and the Taylor-Green initial conditions show peaks in $(\pm 1, 1)$, that is with $\kappa^2 = 1$; the exact solution contains $e^{-2\nu t}$, and it can be seen Section 4.4.6 that:

$$\nu = \frac{\mathcal{LU}}{Re_t} = \kappa_f^{-1} \left(\frac{P}{\kappa_f} \right)^{1/3} \frac{(\kappa_f/\kappa_0)^{4/3}}{Re} = \frac{1}{Re}$$

so the error will always be extremely low if the program has the correct position of the peaks.

The semi-implicit treatment however has a relatively high error with a low number of modes, several orders of magnitude bigger than the exact treatment error. It' necessary to employ at least 150 modes in each non-Hermitian direction to obtain a similar level of accuracy compared to the exact method.

Fig. 6.5 shows more effectively how the error on the u component changes with the spatial resolution: for $N = 4$ the approximate method is 10000 times less accurate than the exact method, for higher values of N this difference reduces to 10 times or less. Nonetheless, for the test case it's been decided to use the exact treatment.

The graph for the v component is not reported since the error is almost

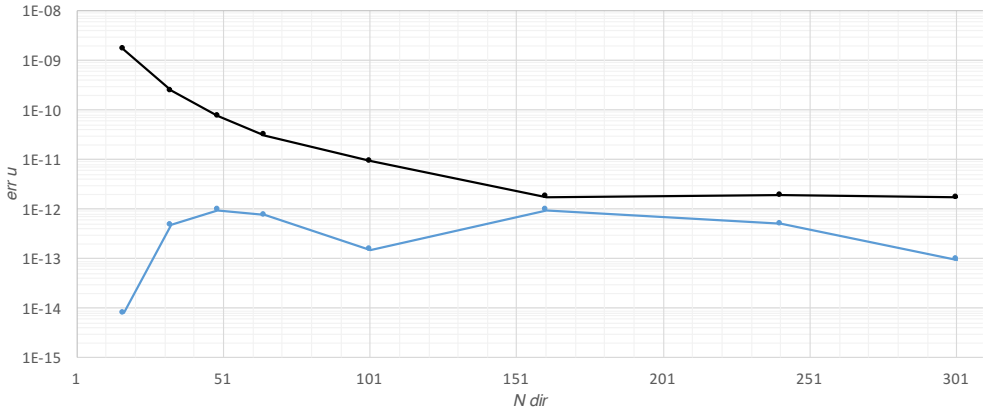


Figure 6.5: Errors on the u component for the Taylor-Green initial conditions; blue = exact method, black = approximate method; N_{dir} is the total number of modes/grid points in a direction ($N_{dir} = 2N + 1$); black = semi-implicit method, blue = exact method.

identical to that of the u component.

Validation of the 3D version

The Taylor-Green analytical solution is only valid in two dimensions. However, since this program solves the problem of homogeneous isotropic turbulence, it is possible to validate the code in three dimensions by rotating the initial condition to obtain the three possible combinations:

$$\begin{cases} u(x,y,0) = sin(x)cos(y) \\ v(x,y,0) = -cos(x)sin(y) \end{cases}$$

$$\begin{cases} u(x,w,0) = sin(x)cos(z) \\ w(x,w,0) = -cos(x)sin(z) \end{cases}$$

$$\begin{cases} v(y,z,0) = sin(y)cos(z) \\ w(y,z,0) = -cos(y)sin(z) \end{cases}$$

Table 6.3 contains the parameters used for testing the 3D version of the program:

The validation results in 3D are contained in Tables 6.4 to 6.6.

The results plotted in Fig. 6.6, where only one component for each of the tree cases is plotted since the other shows almost identical error; the observations for the 3D are similar to the ones for the 2D case. The approximate method shows again that a certain number of modes is required to obtain a comparable level of accuracy. The isotropy od the problem is verified since the error in all three cases is very low.

N	Re	Re_t	$\nu[m^2/s]$	$\Delta t[s]$	$T_{max}[s]$	$\hat{f_\kappa}$	κ_f
8	10	3.969	1.0000000000e-01	0.0005	10	0	2
16	19	7.54	5.2631578947e-02	0.0005	10	0	2
24	28	11.112	3.5714285714e-02	0.0005	10	0	2
32	38	15.08	2.6315789474e-02	0.0005	10	0	2

Table 6.3: Parameters for the validation cases in three dimensions.

N_{dir}	N_{exp}	err_u *exact*	err_v *exact*	err_u *approx*	err_v *approx*
17	32	1.38310e-13	1.38269e-13	1.66916e-09	1.66916e-09
33	64	5.62451e-13	5.62518e-13	2.44896e-10	2.44896e-12
65	128	5.62049e-13	5.62118e-13	3.11074e-11	3.11074e-11

Table 6.4: Errors in the xy plane for an increasing number of modes/points N for two methods of solution; 3D version of the program.

6.3 Forced turbulence

To test the program with a more significant problem, it's been decided to simulate a turbulent forced flow in a periodic box, as is defined in Section 4.4.3, with divergence-free initial conditions (Section 4.4.5).

To decide the minimum simulation time a good it's been decided to follow the indications in [LCP05], where the authors states through a in series of tests it was checked that statistical stationarity can be assumed to start after 10-15 *eddy turnover times*; the number of eddy turnover times that the flow has undergone at time t is defined as:

$$\tau_e = (P\kappa_f^2)^{1/3}t$$

The parameters to set in the program for the simulations are gathered in Table 6.7:

An Adam-Bashforth in two steps (AB2) method is used for the time advancement, while the viscous and forcing terms are solved in the exact manner.

The time step discretisation must be selected carefully in order to keep the Courant number below a certain level during the simulation, in the case of the Adam-Bashforth 2 scheme it must be $C_{max} < 0.2$.

The amplitude of the force-inducing shell is chosen in agreement with [LCP05].

Computing the energy spectrum $E(\kappa)$ during the running time of the program is done by calling the function:

N_{dir}	N_{exp}	err_u *exact*	err_w *exact*	err_u *approx*	err_w *approx*
17	32	1.38546e-13	1.38220e-13	1.66916e-09	1.66916e-09
33	64	5.62555e-13	5.62521e-13	2.44896e-10	2.44896e-10
65	128	7.96796e-13	7.96743e-13	3.11073e-11	3.11073e-11

Table 6.5: Errors in the xz plane for an increasing number of modes/points N for two methods of solution; 3D version of the program.

N_{dir}	N_{exp}	err_v *exact*	err_w *exact*	err_v *approx*	err_w *approx*
17	32	1.38502e-13	1.38527e-13	1.66914e-09	1.66916e-09
33	64	5.62501e-13	5.62546e-13	2.44896e-10	2.44896e-10
65	128	7.96705e-13	7.96864e-13	3.11074e-11	3.11073e-11

Table 6.6: Errors in the yz plane for an increasing number of modes/points N for two methods of solution; 3D version of the program.

```
void energy_spectrum(double time_s);
```

which is called every 1000 iterations and requires the simulation time as input in order to label the saved data. Numerically, the function includes a nested loop that calculates the energy spectrum as:

$$E(\kappa) = \frac{4\pi}{3}\frac{(n+0.5)^3-(n-0.5)^3}{M}\sum_{(n-0.5)\leq|\boldsymbol{\kappa}|\leq(n+.0)}\frac{|\boldsymbol{u}_{\boldsymbol{\kappa}}|^2}{2},$$

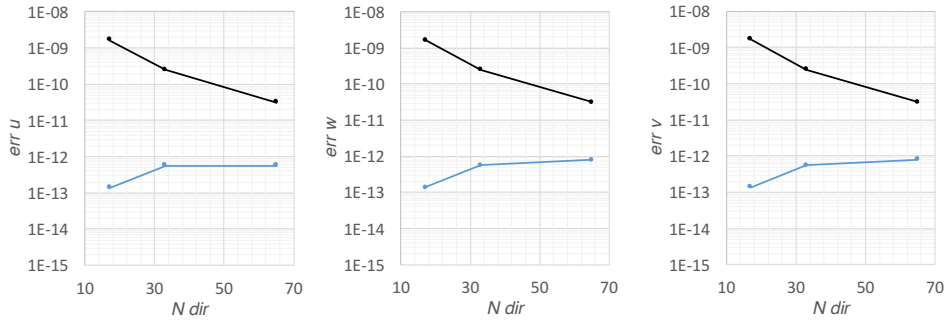
with M being the number of modes contained in the shell centred at $n\kappa_0$.

The computed energy spectra are showed in Fig. 6.7a, along with spectra located in [LCP05]. As was done in the article, the spectra are time-averaged starting from $\tau_e = 10$.

There is good accordance with the spectra obtained by Lamorgese, Caughey and Pope in [LCP05]. The computed spectrum at $Re = 180$ is similar to the reference one at $\kappa_d/\kappa_0 = 50$. The forcing is introduced on the large structures in the energetic region, with a forcing-containing shell of $\kappa_f = 3$; the peak in both images is for $\kappa_f/\kappa_0 < 3$, this confirms the effectiveness of the forcing and the fact that the energy containing scales are in fact the scales of large structures (low wavenumbers).

The compensated energy spectra are represented with a bi-logarithmic plot as a function of κ/κ_d in Figs. 6.8a and 6.8b.

The accordance with reference results is still good, showing that with increasing Reynolds number the computed spectrum approximates the inertial model spectrum for a greater interval of wavenumbers (for more length-



(a) u component for xy **(b)** w component for xz **(c)** v component for yz

Figure 6.6: Errors for some components of the 3D version of the problem compared to the analytical Taylor-Green solution; black = approximate method, blue = exact method.

N_{dir}	N_{exp}	Re	Re_t	κ_d/κ_0	Δt	T_{max}	κ_f/κ_0
85	128	80	18.5	26.8	0.0005	30	3
169	256	180	41.6	49.1	0.0002	30	3

Table 6.7: Parameters for the simulations with forcing defined as in [LCP05].

scales); for Kolmogrov's second similarity hypothesis the separation between dissipative and energetic region increases as the Reynolds number increases.

Figs. 6.9a and 6.9b show respectively the turbulent kinetic energy and the dissipation during the simulation, calculated as in Eq. (3.2) and Eq. (3.3).

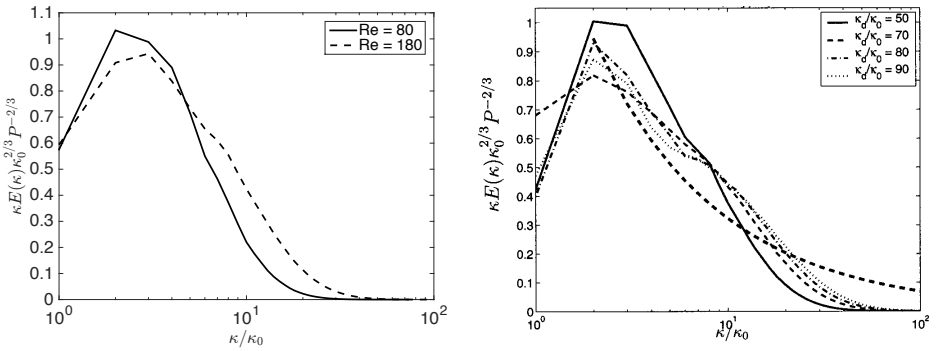
As expected, the dissipation oscillated near the value 1 since it has to compensate $P = 1$ as defined with the forcing scheme used. The turbulent kinetic energy also oscillates near a constant value, indicating that the flow has reached approximately a stationary state.

These graphs show stationary behaviour since $t = 10$ ()the transient is not plot) however the calculated eddy turnover time suggests to use the data starting from $t = 20$ to time-average the results, as it's been done to compute the previous spectra.

Fig. 6.10 shows the overlapping between the energy spectra and the dissipation spectra, computed as:

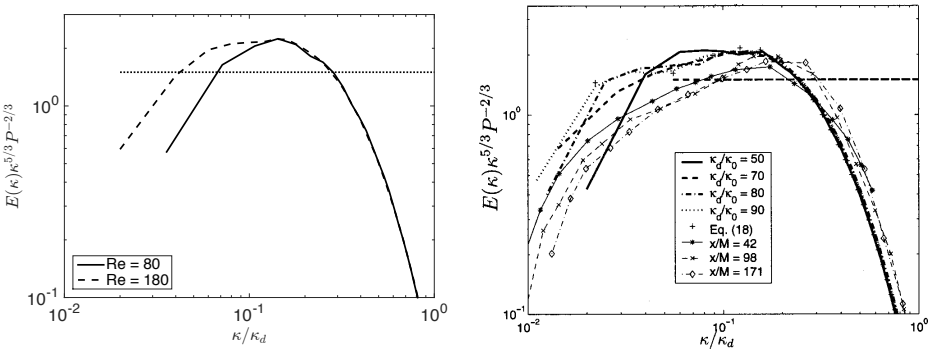
$$D(\kappa) = 2\nu\kappa^2 E(\kappa).$$

As expected, for the low numbers of Reynolds used in this work the energy and dissipation spectra overlap significantly: there is no clear separation of scales. However it can be seen that the case with $Re = 180$ shows greater



(a) computed energy spectra **(b)** reference spectra

Figure 6.7: Energy spectra for the simulated cases and from [LCP05].



(a) computed energy spectra **(b)** reference spectra

Figure 6.8: Energy spectra for the simulated cases and from [LCP05]; the dotted line is the model spectrum for the inertial region.

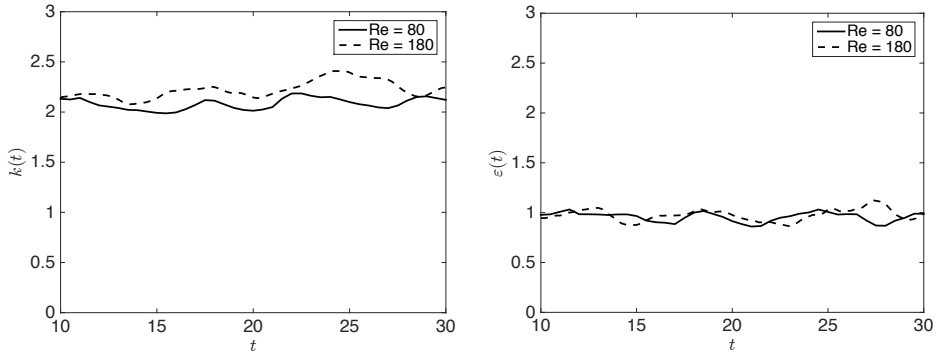
separation compared to the one with $Re = 80$, correctly indicating the trend of more separation with higher Re.

6.4 The hardware

The primary objective of this work is the measurement of the supposed *speed-up* obtainable from the execution of a DNS using the graphical processing unit as an accelerator device.

The baseline CPU execution times have been measure for a similar program developed by Dr.-Ing. Davide Gatti of the Karlsruhe Institute of Technology (KIT), Institute of Fluid Mechanics (ISTM), based on a program created by Dr. Marco Carini.

The above mentioned program is written in the *CPL* programming language developed by Prof. Paolo Luchini and specifically realised to handle DNS problems. For example, this programming language has the ability



(a) computed dissipation spectra **(b)** reference dissipation spectrum

Figure 6.9: Fluctuating quantities during the simulation.

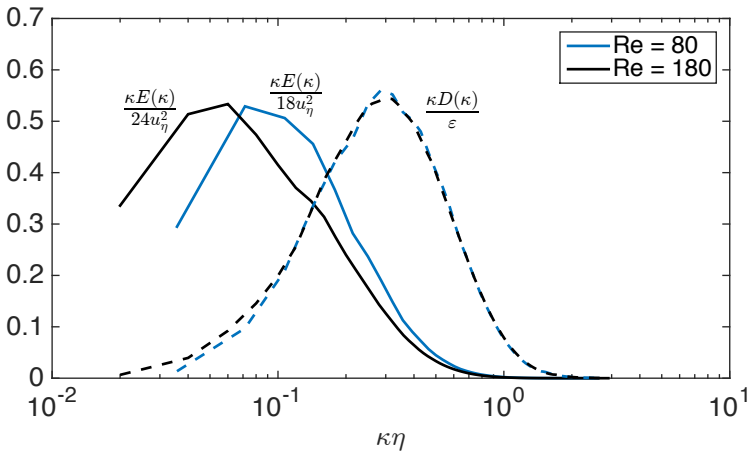


Figure 6.10: Energy and dissipation spectra normalised by the.

to create arrays with negative indices, making particularly easy to access a particular Fourier mode. The CPL toolkit contains a compiler that translate the programs written in this language to highly optimised ANSI C programs. The program used for comparison in this work utilise the high performance FFTW library to execute Fourier transforms.

The CPUs used to measure the above mentioned program are two deca-core Intel Xeon E5-2670v2 at 2.5GHz. Launched in 2013 and based on the *Ivy Bridge-EX* architecture, at default frequency each CPU is capable of 400 Gflops in single precision and 200 Gflops in double precision of peak theoretical processing power.

It was possible to test the developed program on four different graphical processing units; their specifications are in Table 6.8.

It's important to keep in mind that the previously discussed perfor-

name	scope	VRAM	cuda cores	Gflops SP	Gflops DP	frequency [MHz]	cuda capability
GTS 450	desktop	1 GB	192	457	38	1189	2.1
GT 840M	laptop	2 GB	384	790	25	1029	5.0
GTX 780	desktop	3 GB	2304	3977	166	863	3.5
K40c	server	12 GB	2880	4290	1430	745	3.5

Table 6.8: Specifications for the GPUs used in this thesis (SP = single precision, DP=double precision).

mance values measured in *flops* are just theoretical and calculated from the frequency of the CPU/GPU and the number of operations that their particular architecture can execute in one clock cycle.

What can be observed from the previous table is how double precision processing power compares to single precision in different GPU architecture and scope of the product. The GTS450 with *Fermi* architecture has 1/24 the number of double precision execution units per core compared to single precision units, because it is a mainstream product and double precision is only really needed in scientific calculations; the GTX 780 belongs to the *Kepler* family of GPUs but also has the same ratio between double of single precision units.

The GT840M, which belongs to the last GPU architecture *Maxwell* and has an even lower ratio between units: at 1/32, while the GPU is more powerful than the GTS 450 in single precision computation, it is actually about 30% slower in double precision applications.

The Tesla K40c instead is a product developed for the use in server systems, for the exclusive purpose of scientific computations, so it's architecture presents much more double precision execution units compared the other, the ratio is 1/3 in this case.

6.5 Execution times

The 3D version of the program is measured in this work; only the time loop is measured, without initial conditions or post-processing: the influence of the initial condition setting is very low when compared to 200000 executions of the time loop.

The time is measured with the `gettimeofday()` function on Unix systems, which measure wall clock time and not CPU time which is not ideal in this situation (it is the time dedicated by the CPU to a single task); the function is called just before starting the time loop and another just after it is finished.

The CPU version data are provided by Dr. Davide Gatti and measured on the system mentioned above for in the same way.

It was not possible to collect data for all the cases, so the available measured times to execute one time step are in Table 6.9. For the CPU program it was not possible to make more measurements, for the GPU program the data are limited by the amount of memory of each GPU. All the data in the table refers to programs in which the FFTs are executed in-place (IP) except in a series of tests with the Tesla K40c where out-of-place (OOP) mode was used to measure the performance deficit.

	N_{dir}					
hardware	9	21	41	85	169	341
Xeon 1 core	/	/	/	1.32s	16.12s	/
Xeon 4 core	/	/	/	530ms	6.3s	/
Xeon 20 core	/	/	/	/	1.38s	/
GTS 450	0.69ms	1.92ms	12.53ms	99.12ms	/	/
GT 840M	0.82ms	2.8ms	20.99ms	151.4ms	1.164s	/
GTX 780	0.65ms	0.66ms	2.19ms	16.84ms	135.9ms	/
K40c IP	0.52ms	0.53ms	1.61ms	9.93ms	79.73ms	904.6ms
K40c OOP	0.52ms	0.53ms	1.6ms	9.61ms	76.37ms	/

Table 6.9: Time to complete a single time step for the 3D version of the program for different domain dimensions.

While the Nvidia CUDA Programming Guide suggests that in-place transforms are slower, the program is actually only 4-5% slower in in-place mode compared to out-of-place mode. It is a good trade off to be made in order to solve problems with a de-aliased domain of 512^3 using a video card with 12GB of VRAM like the Nvidia Tesla K40c.

Fig. 6.11 represents the results collected in Table 6.9. The first thing noticeable is that doubling the domain dimensions in each direction slows the calculation by approximately a thousand times. The problem is not the velocity field but the non-linear terms structure: the requirement of using a matrix dimension multiple of two for better FFT speed restricts the freedom of choosing the velocity domain dimension. It's the non-linear matrix dimensions that decide the performance; in the 3D case, by doubling the de-aliased dimension the number of elements of that matrix is multiplied by eight: the 3D Fourier transforms dictate the performance (since there are three IFTs and six FFTs in every time step), and with this many elements

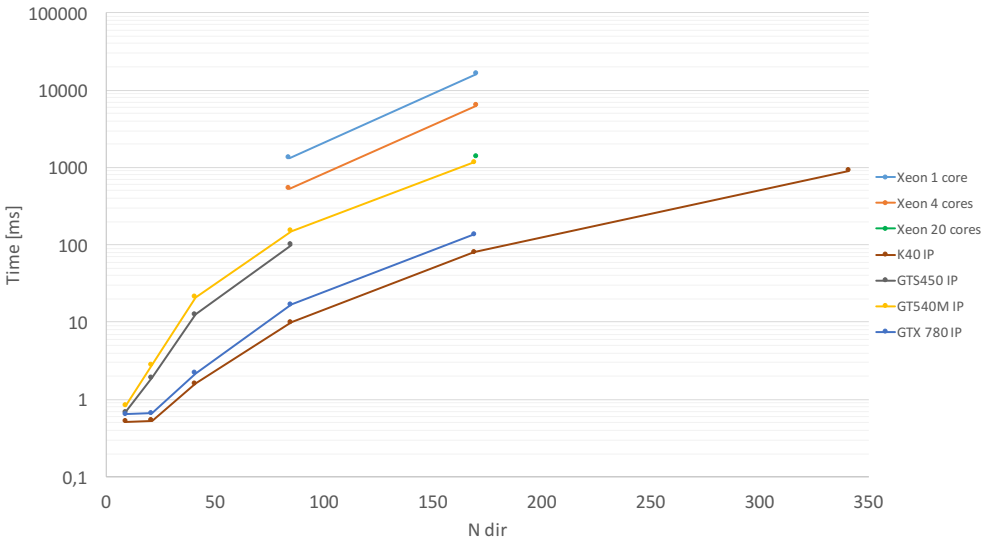


Figure 6.11: Times to execute one time step for different domain dimensions.

to process the performance decreases, as evident in Fig. 6.12.

For a de-aliased domain of 256^3 the GT840M, a mobile chip capable of only 25 Gflops, is as fast as a system with two deca-core processors with 400 Gflops; the GTX780 is about ten times faster than two CPUs. It's clear from these results that it's possible to extract much more performance from a GPU compared to a CPU.

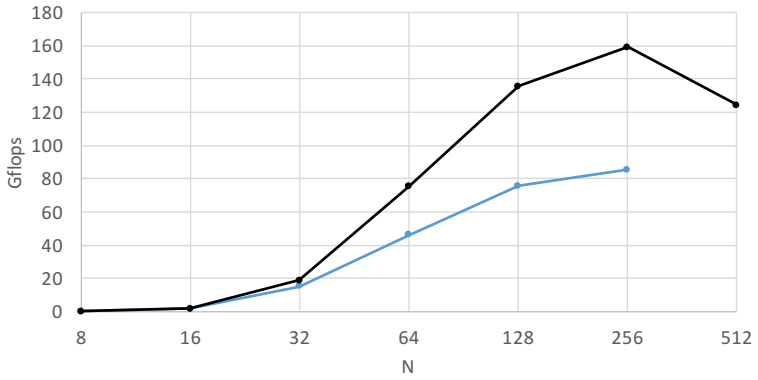


Figure 6.12: Measured performance of an out-of-place real-valued 3D FFT for different transform sizes (N^3); black = Tesla K40c, blue = GeForce GTX 780.

The Tesla K40c, while much more powerful than the GeForce GTX 780 *on paper*, does not show the expected improvements compared to how well the program scales on the other GPUs. The reason for this behaviour is probably the cuFFT library functions which presents a maximum for the

performance they can produce, as it's been measured in Fig. 6.12 with a simple test program containing a real-valued 3D FFT. A 3D FFT is composed of three 1D FFTs that goes on each dimension of the 3D matrix, however the data is aligned only along the third dimension (z) so the memory access pattern is not optimal. To alleviate this situation, various level of cache and other types of memory are used; however, in GPUs the cache is very limited and can't help much in case of big data. These are just guesses, the cuFFT library is closed-source and it's not possible to analyse its functioning.

If the data from the Tesla K40c are removed from the discussion, it's possible to see that the performance scales linearly or better with the increase in theoretical peak performance. For the $N_{dir} = 85$ case, the GTX 780 is 5.8 times faster than the GTS 450 while having only 4.4 times the number of theoretical Gflops; these GPUs have different architectures, with the latter that is newer and certainly better. The GTS 450 is 1.5 times faster than the GT 840M while having 1.52 times the number of theoretical flops.

For the CPU and the $N_{dir} = 169$ case, the improvement from going from one core to four is 2.5×, from one to twenty cores is 11.7×

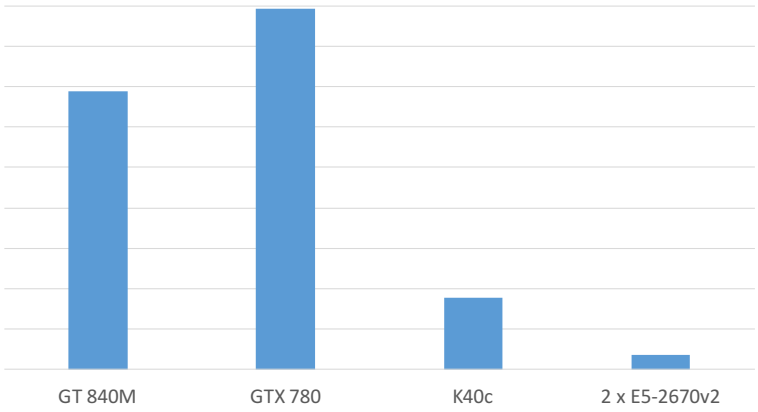


Figure 6.13: For the $N = 169$ (256^3 de-aliased) case, a performance index is calculated as $1/(Gflops * time)$ for the hardware that can make this computation.

Fig. 6.13 contains a representation of a "performance index": this index is supposed to be higher if the execution times are lower ($1/time\_for\_1\_step$), but is also higher if the hardware has lower theoretical peak performance ($1/flops$) in order to reward the cases where this *theoretical* power can be extracted in a bigger part and not just stay on paper. The performance index is the product of these two contributions. This representation shows that the developed program behaves better on the GTX 780, making good

use if the theoretical flops of this video card; the Tesla K40c, for the reasons discussed before, is not able to use its computing power very well: despite being almost nine times faster on paper, the improvement over the GTX 780 in execution times is less than two times. The GT 840M also shows good results with its low theoretical performance of just 25 Gflops. In any case, the GPUs allow to extract more power compared to the theoretical one compared to the high performance CPU used in this comparison

6.6 Energy consumption

One of the reasons why GPUs are used in the most powerful supercomputers is the energy efficiency of these products. While the maximum power that GPUs can absorb is comparable or even higher with respect to modern CPUs, the fact that the work can be completed faster allows to conclude the computation faster and save energy, or consume the same amount of energy but executing more computations.

	2×E5-2670v2	GT 840M	GTX 780	K40c
TDP [W]	2×115	33	250	245

Table 6.10: Thermal Design Power (TDP) for the hardware used in this thesis.

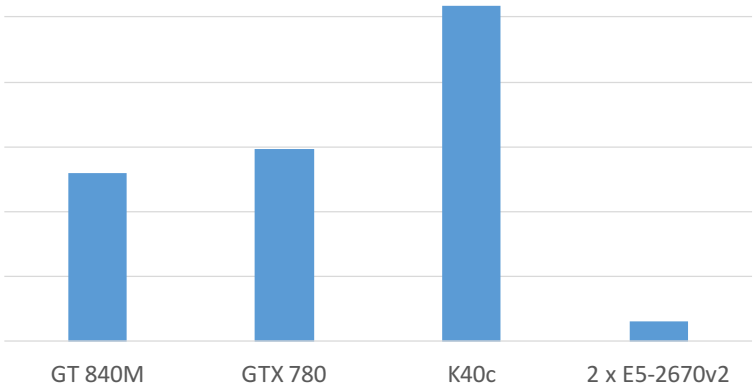


Figure 6.14: For the $N = 169$ (256^3 de-aliased) case, an energy efficiency index is calculated as $1/(TDP * time)$ for the hardware that can make this computation.

Table 6.10 contains informations about the thermal design power (TDP) of the hardware used. While it would have been better to measure the actual energy consumption in each case, the TDP allows to draw some conclusions even if it isn't a precise measurement. For the Tesla K40c it was possible to measure an energy consumption of about 150W, but for fairness this value is not used.

Fig. 6.14 contains a representation of an "efficiency index": this index is higher if the execution times are lower ($1/time\_for\_1\_step$), but is also higher if the hardware uses less power ($1/TDP$) to execute the program. The efficiency index is the product of these two contributions. The determining factor in this graph is how much of the processor is dedicated to double performance computation. Even if the CPU has half the double precision execution unit compared to the single precision ones (unit ratio), its score is very low. While the GT 840M has a unit ratio of just 1/32 and much lower performance compared to the GTX 780 (with a unit ratio of 1/24), the fact that it's a laptop GPU means that energy consumption is very low, about 1/8 of what the desktop GTX 780 consumes.

By looking at the Tesla K40c score it's clear why more and more high performance computers are using GPUs. Even if not fully utilised by the program, it still shows very high performance per watt, about 2/3 better than the GTX 780 and much higher than two Intel Xeon E5-2670v2 CPUs.

From the data of Table 6.9, the GT 840M uses almost the same time to complete a time step for a de-aliased non-linear matrix of 256^3, however the energy consumption to do this computation is about 1/7 of what two deca-core Intel Xeons consume.

7. Conclusions and future developments

In this work a direct numerical simulation code for homogeneous isotropic turbulence has been developed with the purpose of exploiting the computing power of modern GPU architectures.

With the CUDA programming language and toolkit, always evolving and simplifying, the knowledge required to develop a program like this is not excessive if the programmer already has good background of ANSI C, C++, Fortran or Python. There are also features designed to ease the port of applications written in other languages or using other libraries: there is a library with function calls much closed to the ones used with the FFTW library that, with minimal effort, allow to delegate part of the computation to a GPU for an already existing program. Another recent feature added to the CUDA toolkit is *unified memory*, an method of treating memory that automated the transferring of data between host and device memory, thus eliminating explicit memory transfer between these two locations; the performance however are inferior to the manual way, nonetheless it's a good tool for accelerating and simplifying development.

Performance-wise, the results obtained with the developed program are interesting: the graphical processing unit contained in a budget notebook can compete, and even beat, expensive high performance server grade central processing units for this application. By using consumer grade video cards, a desktop computer can accelerate this computation by a factor of 50 times compared to a quad core CPU thanks to a very high performance implementation of the Fast Fourier Transform algorithms in this case. The massive parallelism of modern GPUs makes operating on large data structures fast and efficient, and the different levels of memory included in a GPU allows fast collaboration between different threads.

A conclusion that can be drawn from this work is that, with the acceleration provided by modern GPUs, the limit now is not so much computing times but memory: while it's easy and relatively inexpensive to add more RAM on a computer, a GPU can utilise only the memory it is shipped with. This poses some limitations on what can be done with a single video card: in this case a product with 12GB of VRAM was needed to compute a calculation with a de-aliased domain of 512^3. Using more than one GPU partially solves this problems, but there will always be more memory available to a CPU rather than a GPU.

This computation, DNS of homogeneous isotropic turbulence, was chosen

because is ideal to be parallelised, but it's not always the case. The need for parallel computing depends on whether the type of problems can be parallelised or if they are fundamentally serial calculations. The key to parallel programming is data independence: multiple tasks must be able to operate on data independently. This case presents great data independence, so it was a good candidate to test the performance improvements provided with the massive parallelism of modern GPUs.

Future developments

After a single GPU implementation of this code, the following step could be a multi-GPU version; the CUDA toolkit contains libraries designed to efficiently execute FFTs with data allocated on separated video cards.

Another direction of development could go toward different applications of direct numerical simulations. If multiple GPUs are available, the typical channel flow problems could be divided in horizontal layers and each layers assigned to a different GPU.

The objective is not utilising just the GPU, but to realise an *heterogeneous computation* where the CPU can be tasked with background tasks like intermediate quantities computations, control flow situations or data export while the GPU is occupied with the main computation. The CPU part of the problem could be parallelised across all the cores and used, in this case, to compute the kinetic energy reduction or exporting data to the disk while the main computation is running on the GPU.

This program could be ported to the other major GPU computing programming language: OpenCL. The syntax of this programming language is not too different from the CUDA syntax, so it should not be too difficult. Moreover, it would allow the program to run not only on Nvidia GPUs but also on hardware produced by AMD and Intel. Another tool that could be employed to implement this computation is OpenACC: it is an open source programming standard designed to simplify programming for heterogeneous systems. It stays on a higher level compared to CUDA so it's a bit less efficient: the trade off for simplicity is an inferior ability to fully exploit the hardware.

Translating some kind of programs from CPU code to GPU code can be a difficult task, but the rising utilisation of these devices, the evolving programming tools and the increasing number of professional figures interested in this field makes GPU utilisation every day more convenient.

Bibliography

[Ben15] E. Bendersky. Memory layout of multi-dimensional arrays. `http://eli.thegreenplace.net/2015/memory-layout-of-multi-dimensional-arrays/`, September 2015.

[BT48] G. K. Batchelor and A. A. Townsend. Decay of Turbulence in the Final Period. *Proceedings of the Royal Society of London*, 194(1039):527–543, 1948.

[CGM14] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. Wrox, 2014.

[CT65] J. Cooley and J. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[Dor12] J. Dorrier. The Race to a Billion Operations Per Second: An Exaflop by 2018? `www.singularityhub.com`, 2012.

[FJ14a] M. Frigo and S. G. Johnson. FFTW v3.3.4. `www.fftw.org/`, March 2014.

[FJ14b] M. Frigo and S. G. Johnson. Question 3.11. How can I make FFTW put the origin (zero frequency) at the center of its output? `http://www.fftw.org/faq/section3.html#centerorigin`, March 2014.

[Goo12] N. Goodnight. CUDA/OpenGL Fluid Simulation. `http://docs.nvidia.com/cuda/samples/5_Simulations/fluidsGL/doc/fluidsGL.pdf`, 2012.

[GS66] W. M. Gentleman and G. Sande. Fast Fourier Transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, AFIPS '66 (Fall), pages 563–578, New York, NY, USA, 1966. ACM.

[KAU98] H. Karner, M. Auer, and C. W. Ueberhuber. Top Speed FFTs for FMA Architectures, 1998.

[KH10] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[Kol41] A. N. Kolmogorov. The Local Structure of Turbulence in Incompressible Viscous Fluid for Very Large Reynolds Numbers. *Akademiia Nauk SSSR Doklady*, 30:301–305, 1941.

[LCP05] A. G. Lamorgese, D. A. Caughey, and S. B. Pope. Direct numerical simulation of homogeneous turbulence with hyperviscosity. *Physics of Fluids*, 17(1), 2005.

[LQ06] P. Luchini and M. Quadrio. A Low-cost Parallel Implementation of Direct Numerical Simulation of Wall Turbulence. *J. Comput. Phys.*, 211(2):551–571, January 2006.

[NS14] K. E. Niemeyer and C. J. Sung. Recent Progress and Challenges in Exploiting Graphics Processors in Computational Fluid Dynamics. *J. Supercomput.*, 67(2):528–564, February 2014.

[Nvi15a] Nvidia. Compute Capabilities. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities`, September 2015.

[Nvi15b] Nvidia. cuFFT library user's guide. `http://docs.nvidia.com/cuda/cufft/index.html`, 2015.

[OP72] S. A. Orszag and G. S. Patterson. Numerical Simulation of Three-Dimensional Homogeneous Isotropic Turbulence. *Phys. Rev. Lett.*, 28:76–79, Jan 1972.

[Pop00] S. B. Pope. *Turbulent Flows*. Cambridge University Press, 2nd edition, 2000.

[QL01] M. Quadrio and P. Luchini. A 4th-order-accurate parallel numerical method for the direct simulation of turbulence in rectangular and cylindrical geometries. *15th AIMETA Congress of Theoretical and Applied Mechanics*, 2001.

[Ric22] L. F. Richardson. Weather prediction by numerical process. *Quarterly Journal of the Royal Meteorological Society*, 48(203):282–284, 1922.

[Rog81] R. S. Rogallo. *Numerical experiments in homogeneous turbulence*. NASA technical memorandum. NASA, 1981.

[Sal11] A. Salih. Taylor-Green Vortex, February 2011.

[Sch96] J. C. Schatzman. Accuracy of the Discrete Fourier Transform and the Fast Fourier Transform. *SIAM J. Sci. Comput.*, 17(5):1150–1166, September 1996.

[Sta99] J. Stam. Stable Fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[Tay35] G. I. Taylor. Statistical Theory of Turbulence. *Proceedings of the Royal Society of London*, 151(873):421–444, 1935.

[TG37] G. I. Taylor and A. E. Green. Mechanism of the Production of Small Eddies from Large Ones. *Proceedings of the Royal Society of London*, 158(895):499–521, 1937.

[vK38] L. von Kármán, T.and Howarth. On the Statistical Theory of Isotropic Turbulence. *Proceedings of the Royal Society of London*, 164(917):192–215, 1938.