UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial

Mejora en la eficiencia de un Código de Mecánica de Fluidos Computacional

Trabajo Fin de Máster

Máster Universitario en Ingeniería Aeronáutica

AUTOR/A: Becerra Zúñiga, Nicolás

Tutor/a: Martí Gómez-Aldaraví, Pedro

Cotutor/a externo: MAMORU, TAKAHASHI

CURSO ACADÉMICO: 2022/2023

Agradecimientos

En primer lugar me gustaría empezar dándole las gracias a Takahashi Mamoru, Mukai Toru y Matsui Ryota, los cuales me han ayudado mucho con el proyecto y con mi vida aquí en Japón. Además, dar las gracias a Marcos Carreres y Pedro Martí, mis tutores en la UPV los cuales me han ayudado siempre que lo he necesitado. Y como no a la propia universidad por darme la oportunidad de cumplir un sueño de venir a Japón y desarrollar aquí el final de mis estudios.

A continuación dar las gracias a toda la gente que me ha acompañado este año y medio en Valencia especialmente a Isaac, Raquel y Alex. Ha sido un tiempo breve pero he aprendido mucho y me he sentido muy apoyado por vosotros.

Me gustaría también dar las gracias a mis padres y mi hermano los cuales han hecho posible que sea quien soy y me han brindado la oportunidad de conseguir aquello que siempre he querido.

No me quiero olvidar de mis amigos de Madrid que están guardándome las espaldas siempre que lo he necesitado independientemente de la distancia y del tiempo que haya pasado sin vernos.

Por último gracias a ti Eva, gracias por quererme y apoyarme como lo haces. Esto es el punto final de una etapa y el inicio de una muy emocionante e ilusionante junto a ti.

Sapere Aude.

Resumen

Desde la aparición de la computación a mediados del siglo XX, su aplicación para la resolución de la matemática más avanzada ha ido en aumento. Esto ha permitido la obtención de soluciones precisas que se acercan a la realidad y permiten sacar conclusiones útiles y veraces. No obstante, la limitación de la matemática numérica se encuentra en su propia naturaleza. Pues se basa en la discretización espacial y temporal de los problemas. Este origen, hace que, para un análisis exhaustivo, la nube de puntos a analizar en los problemas pueda alcanzar números exagerados. Esto es sinónimo de un aumento en los tiempos de cálculo relevantes, llegando a alcanzar incluso las semanas. En los últimos años se ha avanzado mucho al respecto, intentando acelerar los cálculos mediante la mejora la memoria de las Unidades Centrales de Procesamiento (CPUs) o con el uso de hardware externo a estas primeras, como es el caso de las Unidades de Procesamiento de Gráficos (GPUs).

Con tal objetivo, el presente Trabajo Final de Máster se centra en la optimización de un Código de mecánica de Fluidos Computacional, el cual consiste en la simulación del flujo turbulento de un fluido entre dos placas inmóviles. Inicialmente se quiere migrar el código de Matlab a lenguaje de programación Julia, el cual es open source, para posteriormente acelerar los cálculos a través de Unidades Gráficas externas. Para finalmente analizar la mejora en la eficiencia y la velocidad de las soluciones. Inicialmente en este estudio se hará un análisis exhaustivo del código Matlab, explicando las ecuaciones de la Mecánica de Fluidos y la metodología numérica utilizada para su resolución. A continuación, se analizará por qué se ha seleccionado Julia como código viable para ello. Después, se hará un análisis sobre la computación con GPUs para finalmente realizar un análisis de los resultados obtenidos a través de estas mejoras (ofreciendo un compromiso entre precisión y coste computacional bajo).

Palabras clave: CFD, Código Julia, GPU, eficiencia, Visual Studio, CUDA

Resum

Des que va aparéixer la computació a mitjan segle XX, s'ha anat aplicant cada vegada més a la indústria per a la resolució de la matemàtica més avançada. Això ha permés l'obtenció de solucions precises que s'acosten a la realitat i permeten traure conclusions útils i veraces. No obstant això, la limitació de la matemàtica numèrica es troba en la seua pròpia naturalesa. Perquè es basa en la *discretización espacial i temporal dels problemes. Aquest origen, fa que, per a una anàlisi exhaustiva, el núvol de punts a analitzar en els problemes puga aconseguir números exagerats. Això és sinònim d'un augment en els temps de càlcul rellevants, arribant a aconseguir fins i tot les setmanes. En els últims anys s'ha avançat molt sobre aquest tema, intentant accelerar els càlculs mitjançant la millora la memòria de les Unitats Centrals de Processament (*CPUs) o amb l'ús de maquinari extern a aquestes primeres, com és el cas de les Unitats de Processament de Gràfics (*GPUs).

Amb tal objectiu, el present Treball Final de Màster se centra en l'optimització d'un Codi de mecànica de Fluids Computacional, el qual consisteix en la simulació del flux turbulent d'un fluid entre dues plaques immòbils. Inicialment es vol migrar el codi de Matlab a llenguatge de programació Julia, el qual és *open *source, per a posteriorment accelerar els càlculs a través d'Unitats Gràfiques externes. Per a finalment analitzar la millora en l'eficiència i la velocitat de les solucions. Inicialment en aquest estudi es farà una anàlisi exhaustiva del codi Matlab, explicant les equacions de la Mecànica de Fluids i la metodologia numèrica utilitzada per a la seua resolució. A continuació, s'analitzarà per què s'ha seleccionat Julia com a codi viable per a això. Després, es farà una anàlisi sobre la computació amb *GPUs per a finalment realitzar una anàlisi dels resultats obtinguts a través d'aquestes millores (oferint un compromís entre precisió i cost computacional baix).

Paraulas clau: CFD, Código Julia, GPU, eficiencia, Visual Studio, CUDA

Abstract

Since the advent of computing in the mid-twentieth century, it has been increasingly applied to industry for solving the most advanced mathematics. This has allowed accurate solutions to be obtained that are close to reality and allow useful and truthful conclusions to be drawn. However, the limitation of numerical mathematics lies in its very nature. It is based on the spatial and temporal discretization of problems. This origin means that, for an exhaustive analysis, the cloud of points to be analyzed in the problems can reach exaggerated numbers. This is synonymous of an increase in the relevant computation times, reaching even weeks. In recent years much progress has been made in this regard, trying to speed up calculations by improving the memory of Central Processing Units (CPUs) or with the use of external hardware, such as Graphics Processing Units (GPUs).

With this objective, the present Master's Thesis focuses on the optimization of a Computational Fluid Mechanics Code, which consists in the simulation of the turbulent flow of a fluid between two immobile plates. Initially we want to migrate the Matlab code to Julia programming language, which is open source, to later accelerate the calculations through external Graphics Units. To finally analyze the improvement in the efficiency and speed of the solutions. Initially, this study will make an exhaustive analysis of the Matlab code, explaining the equations of Fluid Mechanics and the numerical methodology used to solve them. Then, it will be analyzed why Julia has been selected as a viable code for this purpose. Then, an analysis of GPU computing will be made to finally perform an analysis of the results obtained through these improvements (offering a compromise between accuracy and low computational cost).

Keywords: CFD, Julia code, GPU, eficiency, Visual Studio, CUDA

Índice general

Índice de figuras

Índice de cuadros

Glosario de Acrónimos

VARIABLES

CPU	Unidad Central de Procesamiento
GPU	Unidad Gráfica de Procesamiento
HCP	Cálculos de alto rendimiento
$BiCGStab$	Método del Gradiente Bi-conjugado Estable.
LU	Método de factorización Lower-Upper.
CFD	Mecánica de fluidos computacional.
LES	Simulación de torbellinos grandes.
DNS	Resolución numérica directa.
FD	Diferencias finitas.
MIT	masachussets Institute of Technology.
JIT	Programación Just-in-time.
SM	Multiprocesador de streaming.

Capítulo 1

Introducción

En el presente capítulo se introduce el contexto de este *Trabajo Final de Máster* desarrollado en el "Departamento de Ingeniería Mecánica"de la Universidad de MIE (Tsu, Japón), más en concreto en el "Laboratorio de Control de Flujo". Posteriormente se detalla la motivación y los objetivos del estudio finalizando con la explicación de la estructura de este mismo documento.

1.1. Contexto del proyecto

En un principio los problemas del mundo ingenieril se resolvían de manera analítica, descomponiendo el problema en los elementos muy básicos y posteriormente aplicando unas hipótesis que simplificaran aún mas la solución para poder resolverlos a mano. Sin embargo, esto cambió con la aparición de los ordenadores y su consecuente capacidad de resolución numérica. De la misma manera que permitió resolver problemas muchos más complejos, abrió un nuevo abanico de posibilidades, ciencia y retos que afrontar para resolverlos.

Como expuso Alan Turing en su escrito sobre "Computing Machinery and Intelligence" [1] en los años 50, un ordenador consta de tres partes principales:

- **Memoria.** Es la parte en la que el ordenador guarda toda su información. Inicialmente para conseguir unos pocos Gigas de memoria, la logística necesaria ocupaba plantas enteras de edificios. Conforme los años han ido pasando, el problema del espacio físico se ha ido resolviendo. Además se han desarrollado nuevas técnicas con este fin como la aparición de nuevos tipos numéricos conservadores en cuanto al uso de memoria, como es el caso de las matrices dispersas [1].
- **Unidad de ejecución.** Esta parte se encarga de hacer los cálculos. Todo ordenador tiene una Unidad Central de Procesamiento (CPU), la cual, además de realizar los cálculos, controla el tráfico de la información dentro de los diferentes componentes del ordenador. Sin embargo, la velocidad de este tipo de procesador para los cálculos ingenieriles o Cálculos de Alto Rendimiento (HCP) resulta limitada a la hora de realizar muchos cálculos iguales simultáneamente. Es por ello que aparecieron los coprocesadores, como es el caso de las Unidades de Procesamiento Gráfico (GPU), los cuales son capaces de aumentar notablemente la velocidad de los cálculos.
- **Control.** Por último, esta parte es la encargada de corroborar que toda la información que se desea que el ordenador ejecute esté escrita de forma correcta. Antiguamente esta parte, cuyo lenguaje era poco intuitivo y complejo, estaba muy en el "front-end"de la computación, no obstante, con el paso de los años los lenguajes informáticos se han ido sofisticando pasando a ser el "front-end"mientras que el control ha acabado pasado al "back-end"del sistema.

Aunque las tres partes han avanzado notablemente a lo largo de casi 75 años, son problemas que los ingenieros siguen enfrentándose a diario debido a la ambición de resolver problemas

[1] Las matrices dispersas se usan cuando el número de elementos nulos en una matriz supera notablemente al resto de elementos con números. Esto implica que ocupa menos memoria guardar el valor y posición de los elementos en vez de guardar al completo la matriz, como es el caso de las matrices densas

más complejos. En este trabajo se va a trabajar con estos tres partes principales para intentan optimizar la velocidad del código.

1.2. Motivación y justificación del estudio

Como se ha expuesto anteriormente, la capacidad computacional de los ordenadores es cada vez mayor, lo que permite que cálculos más grandes estén al alcance de más personas. Esto ha hecho que la investigación en la dinámica de fluidos computacional (CFD) pueda aplicar técnicas de simulación de flujos turbulentos intensivos en cálculo y con resolución de torbellinos, como las simulaciones "Large Eddy Simulation" (LES) o "Direct Numerical Solution" (DNS).

A partir de esta linea de pensamiento se creó el paquete de software educativo compacto llamado "DNSLab", diseñado para el aprendizaje de ecuaciones diferenciales parciales de turbulencia desde la perspectiva de DNS en el entorno Matlab. Este paquete ha demostrado ser una puerta de entrada muy útil a la simulación de turbulencias para estudiantes de postgrado. Esta implementación en Matlab considera dos de los métodos más comunes de proyección: el método pseudoespectral de Fourier 2D y el método de diferencias finitas 3D con precisión espacial de 2^{o} orden [2], [3] [4]. Este solver simula el desarrollo del flujo turbulento entre dos placas, con un fluido moviendose de forma paralela a ellas.

En el "Laboratorio de Control de Flujos" en el Departamento de Ingeniería Mecánica en la Universidad de MIE, han partido de este paquete para desarrollar sus investigaciones tanto numéricas como experimentales con el fin de estudiar el fenómeno de la transferencia de momento y calor en flujos turbulentos de cizalladura. Pudiendo comparar los resultados posteriormente entre el solver numérico y los experimentos.

1.3. Objetivos del estudio

Este deseo de mejorar el paquete "DNSLab" [5] se ha visto afectado debido al hecho de que Matlab no es un lenguaje "open-source" y a que en caso de que la universidad de MIE pierda la licencia el laboratorio perdería todos sus esfuerzos puestos en esta investigación. Es por esto que han decidido realizar la migración paulatina a, en un principio, un lenguaje de programación más optimizado y "open-source".

La otra limitación que han encontrado es que el tiempo de computación al aumentar la resolución de la simulación numérica crece del orden de $O(N^3)$, siendo N el tamaño de la matriz. Lo que implica que para resoluciones elevadas el tiempo de cálculo deja de ser viable para su investigación.

Este deseo de aumentar la velocidad unido con el interés del laboratorio por el desarrollo de lineas de investigación de Inteligencia Artificial ha hecho que se decidan por investigar cuan efectivo es el uso de coprocesadores externos, en este caso GPUs.

1.4. Estructura del documento

En este informe, inicialmente, se pretende realizar una explicación detallada de la matemática utilizada para llevar a cabo tales objetivos y su implementación computacional en forma de métodos numéricos. Posteriormente se pasará a estudiar los diferentes métodos con los que se busca acelerar el código, centrados principalmente en tres ramas: la implementación en Julia, el uso de precondicionadores y por último el uso de GPUs. Tras ello se explicará el solver y todo el procedimiento de resolución numérica en el paquete. Despues se expondrá como se ha realizado tal implementación en el solver con sus estudios previos de convergencia y velocidad. Por último se analizarán los resultados obtenidos y se expondrán las conclusiones.

Capítulo 2

Fundamentos teóricos

2.1. Ecuaciones de Mecánica de Fluidos Computacional y planteamiento de la solución

2.1.1. Ecuación de Navier-Stokes

El movimiento de un fluido incompresible viene descrita por la ecuación de Navier-Stokes en la ecuación 2.1.

(2.1)

Donde el término convectivo es $\mathbf{C} = \nabla \cdot (\mathbf{uu})$, el gradiente de presión es $\mathbf{P} = -\nabla p$, el término viscoso es $\mathbf{D} = \nu \Delta \mathbf{u}$ y $\mathbf{f}$ describe una fuerza externa. Es importante mencionar que por simplicidad se asume que la presión p es realmente la presión real dividido la densidad constante.

La condición de incompresibilidad, del problema se impone a través de la ecuación de

continuidad (Ecuación 2.2), de esta forma, al ser la densidad constante se obtiene la ecuación 2.3.

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{2.2}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2.3}$$

Aplicando la divergencia en ambos lados de la ecuación de Navier-Stokes (2.1) y teniendo en cuenta que $\nabla \cdot \mathbf{D} = \nabla \cdot \frac{\partial u}{\partial t} = 0$, se obtiene la ecuación de Poisson para la presión.

$$\Delta p = -\nabla \cdot \mathbf{C} \tag{2.4}$$

De este modo, la presión y la velocidad se encuentran acopladas de forma elíptica. Lo cual requiere una solución numérica de un sistema algebraico de ecuaciones.

2.1.2. Método de proyección

Este método desarrollado por Chorin [6] se basa en el teorema de Helmholtz [3] para desacoplar el campo de velocidades del de presiones apoyándose en la ecuación de Poisson (2.4). El teorema de Helmholtz establece que cualquier campo vectorial suave puede expresarse como la suma de dos términos: uno libre de divergencia y otro irrotacional. Por lo que, aplicando esto al problema se puede obtener una velocidad libre de divergencia (u) conociendo una velocidad intermedia suave (u^*) que no satisface incompresibilidad y el gradiente del campo de presiones (∇p) irrotacional.

$$\mathbf{u} = \mathbf{u}^* - \nabla p \tag{2.5}$$

El método de proyección es más sencillo de entender cuando, en el marco de la integración temporal fraccionaria estándar, la derivada temporal es discretizada utilizando el Método de Euler explícito.

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{2.6}$$

En primer lugar la velocidad intermedia u^* es evaluada utilizando solo los términos convectivos $\mathbf{C}^n$, difusivo $\mathbf{D}^n$ y de fuerzas externas $\mathbf{f}_{ext}$ evaluados en el momento temporal "n". Aplicando el método de Euler sobre la discretización temporal Δt, el incremento de la velocidad viene dado por $\partial u = u^* - u^n$, y "u^*" puede expresarse de la siguiente manera:

$$u^* = u^n - \Delta t \left(\mathbf{C}^n - \mathbf{D}^n + \mathbf{f}_{ext}\right) \tag{2.7}$$

En general este campo u^* no está libre de divergencia debido a las no linearidades introducidas por el término convectivo ($\mathbf{C}^n$), por lo que este término tiene que ser proyectado de nuevo a su forma sin divergencia u primero usando la ecuación 2.4 para obtener la presión (p) e introducirla en la ecuación 2.5. De esta manera se obtiene el campo de velocidades del siguiente paso integral el cual cumple con la condición de incompresibilidad.

$$u^{n+1} = u^* - \nabla p \tag{2.8}$$

Es sencillo ampliar el método de Euler utilizado para explicar este procedimiento al método Runge-Kutta de orden 4. En el que la proyección mencionada tiene que hacerse en todos los subpasos para mantener la precisión temporal de cuarto orden del esquema original. Esto además del método de resolución de la ecuación de poisson 2.4 para hallar el campo de presiones (p) será desarrollado con mayor detenimiento en el apartado 2.2.2.

2.2. Métodos Numéricos

Como se ha desarrollado brevemente en la introducción, este trabajo se ha realizado partiendo de una simulación *DNS*, en la que se resuelve de forma directa las ecuaciones

desarrolladas en el apartado anterior. El punto más relevante de este tipo de simulaciones es que la discretización numérica del dominio tiene que ser lo suficientemente buena como para poder resolver todas las escalas espaciales de la turbulencia. Además de esto, al tratarse del estudio de un fluido el cual se desplaza entre dos placas, la capa límite también tiene que ser calculada. Para ello la malla ha sido refinada en la dirección normal a las paredes superior e inferior, de tal manera que el perfil de velocidades de la capa límite pueda desarrollarse libremente captando sus diferentes partes.

Este solver DNS está basado en la resolución mediante Diferencias Finitas (FD). Se trata de un método de carácter general que permite la resolución aproximada de ecuaciones diferenciales definidas en dominios finitos. Este método se basa en la discretización del dominio en una malla de puntos. A partir de estos puntos se es posible evaluar numéricamente las derivadas de la función. Para finalmente poder calcular el comportamiento de estas ecuaciones en todo el dominio de la siguiente manera. Como se ha expuesto anteriormente, en este proyecto se pretende resolver las ecuaciones de Navier-Stokes de forma directa. La ecuación de Navier-Stokes es una ecuación ecuación diferencial parcial respecto al tiempo y a las coordenadas espaciales. Es por ello que cada una de las ecuaciones descritas en la sección 2.1.1 realmente conforman tres, una por cada dirección en el espacio.

El esquema general seguido para la resolución de estas ecuaciones ha sido el siguiente.

1. Discretización espacial: Se divide el dominio en una malla donde se evaluarán las variables del flujo. A partir de esta discretización espacial se calcularán las matrices de derivadas y se impondrán las condiciones de contorno.

2. Discretización temporal: Se define un paso de tiempo Δt y se establece un esquema temporal para avanzar en el tiempo de manera iterativa.

3. Aplicación del método Runge-Kutta. Este método permite obtener la evolución temporal del campo fluido a partir de la concatenación de soluciones para la discreticación temporal anterior. En este caso se desea calcular el estado completo del flujo,

esto implica el cálculo del campo de presiones y no solo de velocidades, por lo que requiere el uso de métodos de cálculo de álgebra lineal, comoe es le método del Gradiente Bi-conjugado EStable (BiCGStab)

4. Iteración en el tiempo. Se repite el paso anterior sucesivamente para avanzar en el tiempo hasta alcanzar el tiempo final deseado o hasta que se cumpla el criterio de convergencia.

2.2.1. Matrices de Derivadas y Condiciones de Contorno

Una manera estándar y compacta de evaluar numéricamente las derivadas es expresar los datos puntuales discretos de la función u como un vector columna $\overline{u}$, y expresar los operadores de derivación, por ejemplo $\frac{\partial u}{\partial x}$, como una matriz de derivadas $\mathbf{D}_x$ [7]. De manera que se puede calcular $\frac{\partial u}{\partial x} \approx \mathbf{D}_x \overline{u}$ de manera muy directa y simple.

De aquí se puede concluir que lo necesario para resolver de forma compacta las ecuaciones de Navier-Stokes es entender qué esquemas de discretización dan soluciones estables, entender cómo y dónde se dan la BC y formar las matrices de derivadas. Tras esto, las ecuaciones continuas pueden transformarse de forma compacta en álgebra lineal.

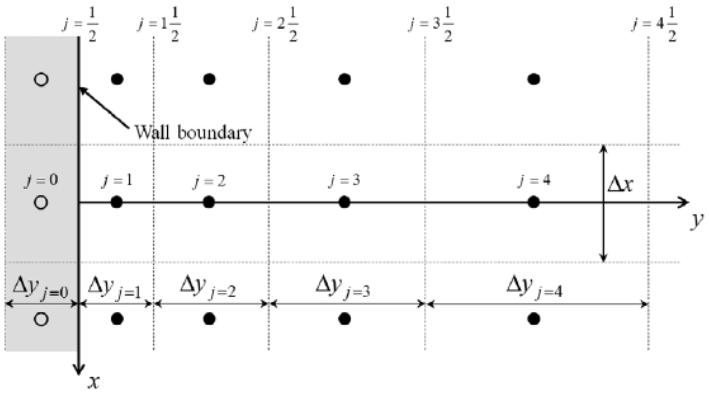


Figura 2.1. Esquema de la malla no uniforme, centrada en las celdas en el plano xy. [8]

En la figura 2.1 se puede observar el esquema de la malla no uniforme cerca de la pared del canal en la dirección normal de la pared. El área blanca representa el interior del canal, mientras que el área gris está fuera de la pared. La pared se encuentra exactamente en la linea de la rejilla en la posición del índice $j = \frac{1}{2}$. La condición de contorno se establece implícitamente en las celdas grises (celdas fantasma) de tal manera que se asigna una condición de contorno deseada exactamente en la ubicación de índice $j = \frac{1}{2}$ como interpolante lineal desde las ubicaciones $j = 0$ y $j = 1$. Para obtener estas matrices se ha utilizado la formulación expuesta por Hirsch [9], expuesta a continuación.

Primera derivada

La primera derivada se evalúa por medias ponderadas, de forma que los valores del campo se interpolan primero a las posiciones del índice $j \pm \frac{1}{2}$ en las caras de la celda, y la derivada en la celda de cálculo se evalúa entonces basándose en estos interpolantes.

$$\left(\frac{\partial u}{\partial x}\right)_j \approx \frac{u_{j+\frac{1}{2}} - u_{j-\frac{1}{2}}}{\Delta x_j} \tag{2.9}$$

Donde los interpolantes $u_{j+\frac{1}{2}}$ y $u_{j-\frac{1}{2}}$ se definen a partir del esquema 2.2.

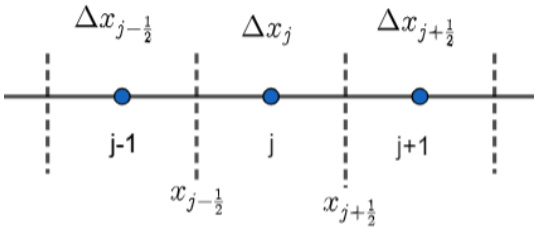


Figura 2.2. Esquema de las variables de interpolación

$$u_{j+\frac{1}{2}} = \frac{\Delta x_{j+1} u_j + \Delta x_j u_{j+1}}{\Delta x_j + \Delta x_{j+1}} \quad , \quad u_{j-\frac{1}{2}} = \frac{\Delta x_j u_{j-1} + \Delta x_{j-1} u_j}{\Delta x_{j-1} + \Delta x_j} \tag{2.10}$$

En cuanto a las condiciones de contorno de Dirichlet, estas hay que imponerlas en la matriz en aquellas mediante el uso de las celdas fantasma ubicadas en los extremos del dominio. La derivada se evalúa de la misma forma, sin embargo el interpolante de la celda fantasma es diferente, pues es donde se aplica la condición de no deslizamiento 2.11. En caso de que deseemos una pared móvil para desarrollar un problema de Couette, solo se tendría que modificar esta ecuación y despejarla para la velocidad de la pared deseada.

$$u_{j-\frac{1}{2}} = \frac{1}{2}(u_j + u_{j-1}) = 0 \Rightarrow u_{j-1} = -u_i \tag{2.11}$$

De esta forma la nueva forma del interpolante $u_{j-\frac{1}{2}}$, el cual se aplicará en las posiciones de la matriz de derivadas correspondientes a las celdas fantasma del dominio,queda de la siguiente forma.

$$u_{j-\frac{1}{2}} = \frac{\Delta x_{j-1} - \Delta x_j}{\Delta x_{j-1} + \Delta x_j} u_j \tag{2.12}$$

Segunda derivada

De manera similar se pueden calcular las segundas derivadas. Aplicando en este caso las condiciones de contorno de Neumann.

$$\left(\frac{\partial^2 u}{\partial x^2}\right) \approx \frac{1}{\Delta x_j}\left[\left(\frac{\partial u}{\partial x}\right)_{j+\frac{1}{2}} - \left(\frac{\partial u}{\partial x}\right)_{j-\frac{1}{2}}\right] \tag{2.13}$$

$$\left(\frac{\partial u}{\partial x}\right)_{j+\frac{1}{2}} \approx \frac{2(u_{j+1} - u_j)}{\Delta x_j \Delta x_{j+1}} \quad , \quad \left(\frac{\partial u}{\partial x}\right)_{j-\frac{1}{2}} \approx \frac{2(u_j - u_{j-1})}{\Delta x_{j-1} + \Delta x_j} \tag{2.14}$$

2.2.2. Solvers

Método de resolución temporal. Runge-Kutta

El método Runge-Kutta, desarrollado al inicio del siglo XX por Carl Runge y Wilhelm Kutta, es una familia de algoritmos numéricos utilizados para resolver ecuaciones de derivadas ordinarias de la forma genérica $\frac{du}{dx} = f(x, u)$. Aunque existen numerosos métodos

en función del orden de precisión y su consecuente estabilidad y velocidad de convergencia [10]. Aunque inicialmente este programa permitía seleccionar el orden del método de resolución, con el fin de simplificar el solver, en la nueva implementación a Julia solo se ha utilizado el método de Runge-Kutta de cuarto orden. El gran potencial de este orden en específico residen en que gracias a su combinación de sucesivos cálculos de pendientes en distintos puntos, el error de truncamiento inherente a los métodos numéricos se reduce considerablemente respecto al resto de métodos numéricos más simples.

Este método de Runge-Kutta consiste en el cálculo del siguiente paso temporal a partir de la media ponderada de cuatro incrementos, donde cada incremento es el producto del tamaño en el intervalo $h = \Delta t$, y una pendiente estimada especificada por la función f en el lado derecho de la ecuación diferencial. La forma de esta función ponderada es la siguiente:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.15}$$

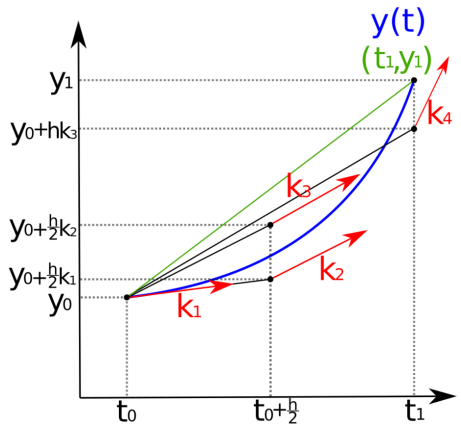


Figura 2.3. Diagrama de las pendientes utilizadas en el método RK4. [11]

Donde

- k_1 es la pendiente al principio del intervalo utilizando y (método de Euler)

- k_2 es la pendiente en la mitad del intervalo utilizando y y k_1
- k_3 es la pendiente en la mitad del intervalo utilizando y y k_2
- k_4 es la pendiente al final del intervalo utilizando y y k_3

Esto se especifica de manera matemática en la ecuación 2.16. Visualmente se puede observar que pendiente se esta calculando en la figura 2.3.

$$\begin{aligned} k_1 &= f(t_n, u_n) \\ k_2 &= f(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}) \\ k_3 &= f(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}) \\ k_4 &= f(t_n + h, u_n + hk_3) \end{aligned} \tag{2.16}$$

Método de resolución del álgebra lineal. BiCGStab.

El método BiCGStab, desarrollado por van der Vorst en los 90, especificamente para la solución numérica de sistemas lineales no simétricos. Se trata de una variante del método del gradiente biconjugado (BiCG) que tiene una convergencia más rápida y suave que este. La principal diferencia del BiCG original es que no requiere multipliación por la traspuesta del sistema de la matriz.

2.3. Métodos de aceleración numérica

En este apartado se van a exponer las tres principales iniciativas que se han planteado con el fin de acelerar el solver numérico de CFD. La primera es independiente del código utilizado: los precondicionadores. Se trata de un recurso de métodos numéricos que facilitan el cálculo al tratar sistemas lineales con matrices esparcidas muy grandes. El segundo método es la migración del código de Matlab a otro código, en este caso Julia. Se expondrán las razones por las que se ha seleccionado y posteriormente se analizarán si realmente se ha obtenido el

resultado deseado. La última iniciativa es la aceleración mediante procesadores externos, en este caso GPUs. Estos procesadores están especializados debidos a su naturaleza a hacer muchos cálculos simples de forma simultánea.

2.3.1. Precondicionadores

Los precondicionadores son un tipo de arreglo matemático usado en el álgebra lineal que tiene como fin aumentar la tasa de convergencia del cálculo numérico. Póngase por caso que se tiene un sistema lineal de la forma $\mathbf{Ax} = \mathbf{b}$ siendo $\mathbf{A}$ una matriz de nxn y $\mathbf{b}$ y $\mathbf{x}$ vectores de dimensión n, su convergencia se puede ver muy aumentada si se aplica el método de resolución iterativo a un sistema precondicionado equivalente:

$$\mathbf{M}^{-1}\mathbf{A}x = \mathbf{M}^{-1}\mathbf{b} \tag{2.17}$$

La matriz de precondicionamiento no singular M debe ser una buena aproximación a $\mathbf{A}$ y fácil de calcular, los sistemas de ecuaciones obtenidos deben resolverse eficazmente en operaciones de orden n y, además, $\mathbf{M}$ debe tener unos requisitos bajos de almacenamiento. Con el producto $\mathbf{M}^{-1}\mathbf{A}$ se busca obtener una matriz más cercana a una matriz diagonal o una matriz más fácil de manejar que la original. Pues las matrices diagonales, o con valores entorno a la diagonal, asegura una mayor independencia de las componentes, facilita la inversión en caso de que sea necesaria y simplifica el comportamiento del sistema, pues la solución para una variable no depende fuertemente de otras,lo que facilita la convergencia. Una de las opciones más populares para $\mathbf{M}$ es la factorización LU incompleta, y es la que ha sido implementada en el código de julia.

Factorización LU incompleta

Esta factorización LU consiste en el cálculo de una matriz triangular inferior dispersa "**L**" y una matriz triangular superior dispersa "**U**" de forma que la matriz residual $\mathbf{R} = \mathbf{LU} - \mathbf{A}$ satisfaga ciertas restricciones, como tener entradas nulas en algunas posiciones. Básicamente, aunque la multiplicación de ambas matrices no sea igual a la original, las posiciones que tienen que ser cero se impone esta condición, de ahí que se la conozca como incompleta. Esta aproximación de las matrices factorizadas permite una reducción significativa del almacenamiento y de los cálculos necesarios.

De esta manera el sistema original se puede reestructurar de la siguiente manera.

$$\mathbf{Ly} = \mathbf{b} \tag{2.18}$$

con $\mathbf{y} = \mathbf{Ux}$. Esta forma hace que la resolución que antes era directa y computacionalmente cara se divida en dos operaciones que se calculen de manera muy rápida y eficaz.

2.3.2. Migración de código

A pesar de que la principal razón de la migración del código es económica, se ha decidido intentar acelerar el código a través de esta migración, debido a que realizarla era inevitable.

Inicialmente se consideraron dos lenguajes: Python y Julia. A continuación se van a exponer las principales diferencias de ambos lenguajes. El análisis se va a centrar en 4 puntos, a continuación expuestos:

- **Diseño y enfoque:**
  - Python: Es un lenguaje de programación generalista, diseñado para ser fácil de leer y escribir, con una amplia comunidad y una extensa biblioteca estándar que abarca muchas áreas de desarrollo.

  - Julia: Es un lenguaje creado específicamente para el cómputo científico y numérico de alto rendimiento. Pretende ser rápido como C y fácil de usar como Python. Su desarrollo se inició en 2009 en el MIT, pero no fue hasta 2018 que se consideró un lenguaje estable.

  Puesto que este programa es de cálculo ingenieril, se puede afirmar que debido a que Julia está más enfocado cálculo científico es mejor para su implementación.

- **Tipado:** Ambos son lenguajes de tipado dinámico, es decir, su tipo es asignado por el intérprete durante el tiempo de ejecución basado en su valor en ese momento. Aunque Julia si que hace esta asignación opcional, en Python esto se hizo posible a partir de la versión 3.5.

- **Rendimiento:**

  - Python: Es un lenguaje interpretado, lo que puede llevar a un rendimiento más lento en comparación con lenguajes compilados (como Julia). No obstante, se pueden utilizar extensiones y bibliotecas en lenguajes de bajo nivel para mejorar el rendimiento en tareas específicas.
  - Julia: Ha sido diseñado para ofrecer un alto rendimiento. Gracias a su compilador Just-In-Time (JIT), puede igualar o incluso superar el rendimiento de lenguajes tradicionalmente más rápidos, como C o Fortran, en ciertos escenarios.

- **Sintaxis:** Tanto Python como Julia son lenguajes con sintaxis muy sencilla por lo que la lectura de su código es muy sencilla en ambos casos.

- **Manipulación de matrices y arreglos:** Este es, precisamente, el punto fuerte de Julia. Está especificamente pensado para que la manipulación sea rápida y eficaz. En el caso de Python, su rendimiento en cálculos intensivos no es tan bueno como Julia. Para cálculos científicos se suelen utilizar bibliotecas como NumPy [12] y SciPy [13].

- **Madurez de comunidad y ecosistema:** Puesto que Python es un lenguaje mucho más genérico y desarrollado, su comunidad es mayor y por ende la cantidad

de librerías accesibles es notablemente superior. Sin embargo, debido a que Julia se especializa en la manipulación de arreglos matemáticos, sí que existen librerías desarrolladas y optimizadas para ciertos cálculos.

- **Uso de GPUs** Para poder utilizar unidades de procesamiento gráfico de NVIDIA hay diferentes maneras de hacerlo. Tanto Julia como Python permiten el uso de GPUs a un nivel alto, es decir, para desarrolladores que no están especializados en este tipo de manejos de memoria. Por el hecho de que la madurez del ecosistema es mayor en Python, se puede pensar que este puede estar por encima de Julia, sin embargo es lo contrario, debido al mismo motivo que se ha expuesto con el caso anterior [14], Julia está muy enfocado en el cálculo de alto nivel de computación y la librería que permite el uso de GPUs (CUDA [15]) está muy optimizada. En el siguiente punto se entrará más en detalle en el caso de Julia.

Debido al inherente enfoque de cómputo científico, sumando otros factores como facilidad del código, un alto rendimiento a la hora de tratar con grandes arreglos matemáticos y con la existencia de CUDA.jl, una librería que permite el uso de GPUs en un nivel alto de programación se ha decidido hacer la migración a Julia. A continuación se va a entrar un poco más en detalle en este lenguaje.

Julia

Julia es un lenguaje de programación de alto nivel de abstracción, dinámico y orientado para el análisis numérico y la ciencia computacional.

Debido al aumento de la producción de programas de cálculo numérico, los lenguajes de alto nivel de abstracción ha ido en aumento (Python, Matlab, Julia... etc). Esto se debe a que en su propia definición va implícito una simple implementación del código por parte del usuario, alejado del lenguaje interpretado por la máquina. Además, al ser dinámico, permite al usuario no prestar tanta atención a los tipos de las variables, las cuales pueden

ir modificándose a lo largo del código, a costa del rendimiento del mismo.

Julia está enfocado sobre todo hacia la alta eficiencia. Mezclando los tipos de variables dinámicos en la ejecución y un compilador JIT (Just-In-Time) utilizando a su vez la infraestructura del compilador LLVM, permite generar codigo máquina de forma altamente eficiente.

2.3.3. Uso de GPUs

A finales de los 80 aparecieron este nuevo tipo de procesadores cuya principal función era aligerar carga de trabajo de la CPU en aplicaciones como videojuegos o aplicaciones 3D interactivas. La gran ventaja de las GPU es que están altamente segmentadas lo que permite realizar una elevada cantidad de operaciones sencillas iguales de forma simultanea. Mientras que la CPU tiene una forma de funcionar secuencial en un único hilo de computación (Thread), salvo que se especifique lo contrario. Y, aunque se programe la CPU de tal manera que sea capaz de realizar multitareas, la cantidad de Threads que puede soportar es notablemente inferior a la que se puede obtener con una GPU.

En los últimos años el desarrollo de las GPU ha dado un crecimiento increíble, no solo por los videojuegos, si no también por el mundo de las criptomonedas, la ingeniería y de la Inteligencia Artificial. Esto se puede observar claramente en la empresa de NVIDIA que desde 2016, el precio de sus acciones se ha disparado (Figura 2.4). Esta empresa es la más grande a nivel mundial distribuidora de GPUs.

Figura 2.4. Evolución de las acciones de NVIDIA desde su creación en 1998. [16]

Aunque la estructura de las GPUs depende de los fabricantes todos se basan en el mismo principio: tener muchos procesadores que ejecuten cálculos sencillos de manera simultanea.

En el caso más concreto de Nvidia, los diferentes threads, hilos de cálculo, se agrupan formando bloques que a su vez se unen formando redes de kernels[1]. Esta estructura computacional tiene su forma física, cada thread es un core, que se agrupa formando Streaming Multiprocessors (SM), que a su vez conforma la GPU en su conjunto.

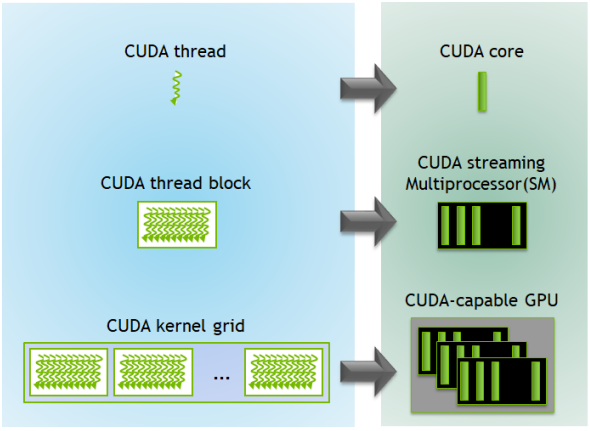


Figura 2.5. Ejecución de kernel en GPU. [17]

[1]Un CUDA Kernel es una función que se ejecuta en la GPU

Inicialmente para este proyecto se utilizó una GPU Quadro 4000, no obstante el laboratorio al contar con más financiación, se decidió comprar una nueva. Para ello se hizo un análisis de mercado para ver qué unidad comprar dado un presupuesto de unos 2000 euros.

La primera especificación que se tuvo en cuenta fue la marca de la nueva GPU. Aunque las gráficas AMD son muy buenas relación calidad-precio, se decidió que se deseaba adquirir una Nvidia. Esto se debe a que, las gráficas Nvidia tienen un mayor ecosistema de implementación en todo tipo del lenguaje, haciendo sencillo su manejo para desarrolladores noveles. Otro requerimiento que marcará la decisión final es que se priorizará el uso de redes neuronales, por encima de los cálculos de alta computación (HCP). Esto hace que la potencia de precisión simple prime sobre la precisión doble. Sin embargo, este análisis se realizó en base únicamente de la potencia de cálculo de precisión doble.

De esta búsqueda se extrajo la tabla 2.1.

GPU	Arquitectura	FP64 Tasa de rendimiento	Precio (€)
Nvidia A100	Ampere	9.7 TFLOPS	15000
Quadro GV100	Volta	8.3 TFLOPS	6000
Nvidia Titan V	Volta	7.8 TFLOPS	3800
Nvidia Tesla V100	Volta	7.8 TFLOPS	3800
Nvidia Tesla V100s	Volta	7.4 TFLOPS	2250
Nvidia A30	Ampere	5.16 TFLOPS	4500

Tabla 2.1. Diferentes GPUs en el mercado

Lamentablemente todas las encontradas superaban el límite de los 2000 euros de presupuesto, sin embargo, la más próxima a esta sería la Nvidia Tesla V100s. Finalmente, la GPU utilizada para este estudio ha sido la RTX A5500 [18]. Una GPU de ultima generación (2022) con arquitectura Ampere, 24Gb de memoria y un rendimiento teórico de para cálculos de tipo doble precisión (FP64) de 1.066 GFlops. Este rendimiento no es muy elevado, ya que existen otros modelos similares como la NVIDIA TESLA V100 con

un rendimiento de 7 TFlops, un 85 % más. Este modelo se ha utilizado debido a razones ajenas a este proyecto, pues está especializado en tipos de precisión simple, los usados por las inteligencias artificiales.

Capítulo 3

Definición del Problema

En este capítulo se va a exponer el caso que se ha analizado y las características del proceso de su solución.

3.1. Configuración del caso

El flujo del canal es impulsado en la dirección x por una fuerza externa constante con el fin de conseguir una situación de flujo turbulento estacionario. Por otro lado, en las paredes superior e inferior se cumple la condición de no deslizamiento, asegurando así que la velocidad del fluido es 0 en la pared. En cuanto a las paredes de la dirección de la corriente y perpendiculares a esta se imponen condiciones de contorno periódicas.

Es bien conocido que la situación de un flujo turbulento estacionario se alcanza cuando el esfuerzo cortante de la pared y la fuerza del cuerpo se equilibran mutuamente[3] [5]. Dicho equilibrio da lugar a una relación simple entre el número de Reynolds , la fuerza externa $\mathbf{f}_x$, la altura del canal H y la viscosidad (Ecuación 3.1).

$$\mathbf{f}_x = \frac{8\nu^2}{H^3} Re_\tau^2 \tag{3.1}$$

En este estudio se fija $\mathbf{f}_x$ de tal forma que $Re_\tau^2 = 180$ corresponde a un caso muy bien de baja turbulencia muy bien establecido que proporciona un buen problema de prueba para la evaluación comparativa del solver de diferencias finitas [5] .

El dominio de esta simulación tiene forma ortoédrica y las dimensiones vienen recogidas en la siguiente tabla:

Largo	Ancho	Alto
2π	π	2

Tabla 3.1. Dimensiones nominales del dominio

Condiciones de contorno: no slip + simetría

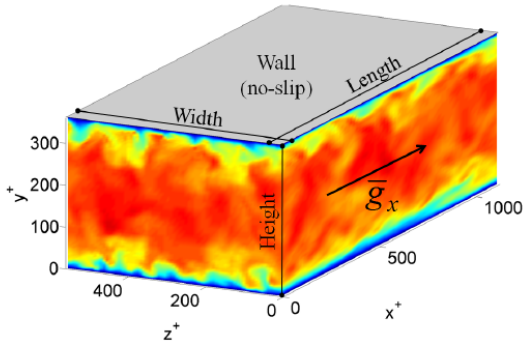
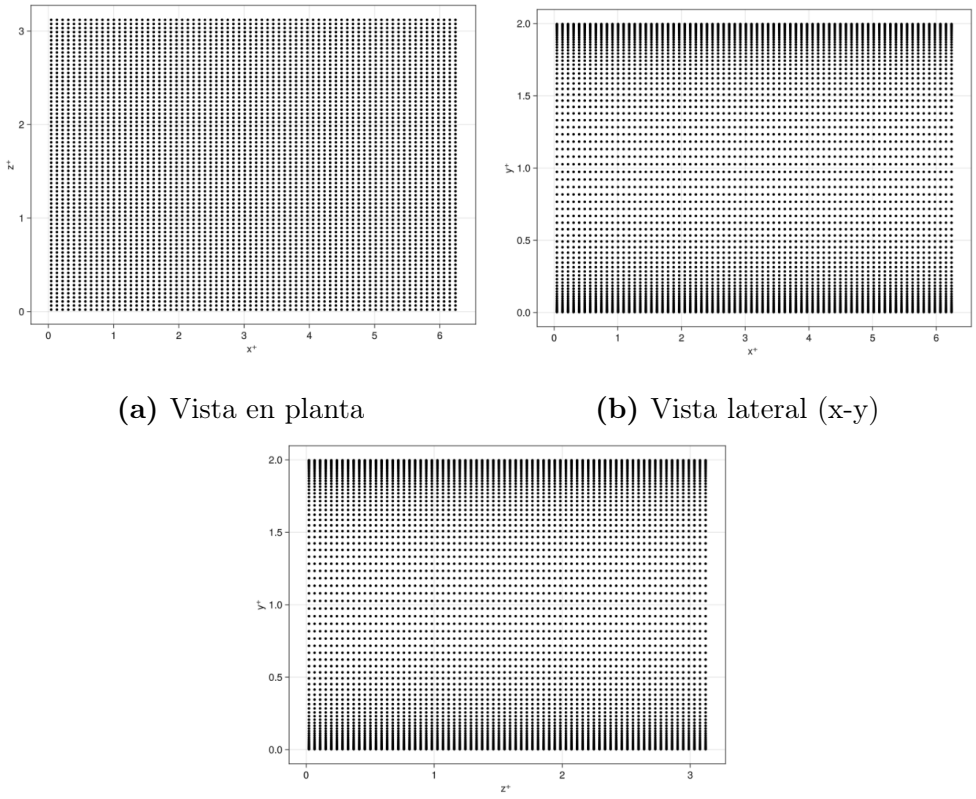


Figura 3.1. Esquema del dominio del problema.[8]

3.2. Malla

El mallado del dominio se realiza a partir de la definición de la resolución del problema, asociado a la variable "N". Este número indica el número de puntos en el que se discretiza cada dirección del espacio. La malla se genera de forma individual según cada dirección. Para las direcciones x^+ y z^+ esta discretización es uniforme. Mientras que para la dirección

y^+, normal a las paredes, se utiliza una discretización según la tangente hiperbólica. De tal manera que las celdas de la malla disminuyen de tamaño al acercarse a las paredes, como puede observarse en la imagen 3.2. Este mecanismo de refinamiento se controla a través de un parámetro (”ctanh”), el cual cuanto menor sea, menor será el refinamiento hacia los extremos.



(a) Vista en planta **(b)** Vista lateral (x-y)

(c) Vista frontal (z-y)

Figura 3.2. Malla del dominio

3.3. Matrices de diferencias

Para el cálculo de la velocidad se requieren un total de 6 matrices de diferencias: D_x, D_y, D_z, D_xx, D_yy y D_zz. Para el cálculo de la presión también se requieren otras 6: D_x^p, D_y^p, D_z^p, D_{xx}^p, D_{yy}^p y D_{zz}^p. El tamaño de estas matrices es de $N^3 \times N^3$. Tanto para la presión

como la velocidad, las condiciones de contorno son periódicas en las direcciones x y z. Mientras que en la dirección y, la normal a las paredes las condiciones de Dirichlet o Neumann tienen que ser implementadas como ya se ha explicado en la sección 2.2.1. Estas matrices solo se crean al inicio de la simulación, pues la malla no se modifica a lo largo de la simulación.

3.4. Evaluación del incremento de la velocidad

Los operadores del campo de velocidades actualmente están representados como matrices esparcidas de $N^3 \times N^3$. Para calcular las derivadas del campo de velocidades es necesario almacenar estos campos en vectores columna de tamaño $N^3 \times 1$ ($\bar{u}_n$, $\bar{v}_n$, $\bar{w}_n$). En esta forma matriz-vector, el término convectivo para la ecuación del momento según la dirección x puede ser evaluado de forma antisimétrica como a continuación:

$$\bar{C}_x = \frac{1}{2}[D_x(\bar{u}_n\bar{u}_n) + D_y^p(\bar{v}_n\bar{u}_n) + D_z(\bar{v}_n\bar{u}_n) + \bar{u}_n(D_x\bar{u}_n) + \bar{v}_n(D_y\bar{u}_n) + \bar{w}_n(D_z\bar{u}_n)] \quad (3.2)$$

Donde los productos del tipo $\bar{u}_n\bar{u}_n$ indican multiplicación termino a término. Cabe destacar que por razones de consistencia con la definición de la condición de contorno de no deslizamiento la matriz D_y^p se aplica sobre el producto de velocidades en la dirección y, normal a las paredes. De forma similar se calcula el término difusivo en la dirección x como:

$$\bar{D}_x = \nu D_{xx}\bar{u}_n + \nu D_{yy}\bar{u}_n + \nu D_{zz}\bar{u}_n \quad (3.3)$$

De esta manera el incremento de la velocidad en los centroides se puede calcular como la suma de ambos términos:

$$\begin{aligned} \partial\bar{u} &= \Delta t(-\bar{C}_x + \bar{D}_x + f_x) \\ \partial\bar{v} &= \Delta t(-\bar{C}_y + \bar{D}_y) \\ \partial\bar{w} &= \Delta t(-\bar{C}_z + \bar{D}_z) \end{aligned} \quad (3.4)$$

3.5. Proyección

La ecuación de Poisson (Ecuación 2.4) se puede discretizar en un modelo compacto a través del uso de la matriz de derivadas de segundo orden, explicada a través de la ecuación 2.13, en las direcciones de la corriente y perpendicular a esta. Por otro lado, la condición dde contorno de Neumann $(\frac{\partial p}{\partial y})_{wall} = 0$ se prescribe en las paredes para la presión [19]. En una malla uniforme bastaría con aplicar al modelo estandar de siete puntos, en el cual el operador laplaciano se opera a partir de los 6 puntos entorno al punto a calcular más el punto en el que se va a calcular el operador Laplaciano. Sin embargo, en las caras de las celdas en contacto con la pared, la condición de contorno de Dirichlet es impuesta a la presión de referencia ($p = 0$) con el fin de hacer la solución de la ecuación de Poisson única.

A partir de lo explicado anteriormente es posible crear la matriz de Poisson (M). Puesto que la malla no utiliza nodos fantasma, las condiciones de contorno se implementan en la diagonal de la matriz M, resultando en un modelo de 6 puntos en las paredes. Los elementos de la matriz también son modificados en aquellas celdas donde las condiciones de contorno Dirichlet para imponer la presión de referencia. Como resultado, el operador de Poisson para este problema queda definido como la suma de 3 matrices

$$M = D^p_{xx} + D^p_{yy} + D^p_{zz} \tag{3.5}$$

Una vez las velocidades intermedias, $\bar{u}^*$,$\bar{v}^*$,$\bar{w}^*$, han sido actualizadas el campo de presiones se resuelve aplicando la ecuación de Poisson:

$$M\bar{p} = D_x\bar{u}^* + D_y\bar{v}^* + \bar{w}^* \tag{3.6}$$

Esta ecuación se resuelve a través del método estable del gradiente Bi-conjugado precondicionado (BiCGStab), con un precondicionado incompleto LU (ILU). Tanto la matriz M como el precondicionador son generados antes de que la simulación empiece, por lo que se deben calcular una única vez. Finalmente, una vez la presión es conocida, la proyección

puede llevarse a cabo usando diferencia de matrices:

$$\begin{aligned} \bar{u}_{n+1} &= \bar{u}^* - D_x^p \bar{p} \\ \bar{v}_{n+1} &= \bar{v}^* - D_y^p \bar{p} \\ \bar{w}_{n+1} &= \bar{w}^* - D_z^p \bar{p} \end{aligned} \tag{3.7}$$

Como resultado, el campo $\bar{u}_{n+1}$, $\bar{v}_{n+1}$, $\bar{w}_{n+1}$ no tiene divergencia.

Capítulo 4

Implementación

En este capítulo se analizará como se ha realizado la migración del código de origen Matlab a Julia. Y como se ha ido optimizando este en diferentes partes. Obteniendo dos codigos fuente: uno con cálculo CPU y otro con hardware externo (GPU).

4.1. Librerías utilizadas

Las librerías son conjuntos de archivos de código, usualmente compilado que se utilizan para el desarrollo de software. En este proyecto se han utilizado un total de 10 librerías.

- **CUDA.jl [15]**: Esta librería es la principal interfaz de programación para trabajar con las GPU NVIDIA CUDA. Incluye una abstracción de arreglos fácil de usar, un compilador para escribir kernels CUDA en Julia y "wrappers" para varias librerías CUDA (CUSPARSE, para el manejo de arreglos dispersos).
- **GLMakie.jl [20]**: GLMakie es una librería de visualización de datos para el lenguaje de programación Julia, de alto rendimiento y extensibilidad. Se usa para el ploteo de todo tipo de gráficas y figuras.

- **Printf.jl**: Es una de las librerías estándar de Julia, permite imprimir args usando cadena de especificación de formato estilo printf de C.
- **Linear Operators.jl [21]**: Esta librería permite crear objetos llamados operadores, los cuales se comportan como matrices, pero se definen por su efecto cuando se aplican a un vector. Pueden transponerse, conjugarse o combinarse con otros operadores a bajo coste. Permiten aplazar las operaciones más costosas hasta el momento de multiplicar con un vector.
- **Krylov.jl [22]**: Este paquete proporciona implementaciones de algunos de los métodos Krylov más útiles para una gran variedad de problemas: Sistemas de rango completo cuadrados o rectangulares, problemas de mínimos cuadrados, problemas lineales de norma mínima y sistemas adjuntos. Este paquete permite el uso de la GPU para su cálculo. Este punto es clave a la hora de acelerar el solver.
- **LinearAlgebra.jl**: Es otra de las librerías estándar de Julia, proporciona implementaciones nativas de muchas operaciones de álgebra lineal comunes y útiles.
- **Statistics.jl**: La última librería estándar utilizada de Julia, contiene funciones estadísticas básicas. Esta se ha utilizado para el postprocesado de los resultados obtenidos del campo fluido.

4.2. Código

En este apartado se va a explicar el procedimiento seguido paso a paso para acelerar el código con los diferentes puntos expuestos en el capítulo 2.3.

4.2.1. Implementación a Julia

La implementación a Julia desde Matlab no muy tediosa, ya que ambos lenguajes de programación son muy similares. Sin embargo, en el caso de la creación de las matrices de

derivadas no fue así, pues para el cálculo en GPU hubo que vectorizar el código, haciendo mucho más largo el código y más complejo de entender. Esto se desarrollará con más detenimiento en el apartado 4.2.3.

Además de esto, se decidió simplificar la estructura original del código a lo más simple posible, con el fin de facilitar la legibilidad posterior del mismo. Un par de ejemplos de esto es que el postprocesado perdió importancia y por ello no está tan desarrollado como el original o que el único método de simulación temporal es el método de Runge-Kutta de orden 4 mientras que el original tiene la opción de calcularlo en diferentes órdenes.

Las principales diferencias entre ambos lenguajes se hallaron a la hora de definir ciertos operadores, como bucles y condicionales, o de ”indexar” posiciones específicas de arreglos donde Julia utiliza corchetes mientras que en Matlab se utilizan paréntesis. La última diferencia y la más delicada son las funciones nativas de Matlab.

Matlab al ser un programa ya con muchos años, tiene un ecosistema muy desarrollado y optimizado. Esto hace que tenga numerosas funciones disponibles que en el caso de Julia hay que buscar online, en librerías aceptadas por la comunidad para utilizarlas. Esta situación se ha dado en diferentes puntos de la implementación pero las más importantes fueron:

- Creación de la estructura numérica de la malla. (Meshgrid. Figura 4.1)

```
[X,Y,Z] = meshgrid(x,y,z);
```

(a) En Matlab.

```
X = getindex.(Iterators.product(x, y, z), 1);
Y = getindex.(Iterators.product(x, y, z), 2);
Z = getindex.(Iterators.product(x, y, z), 3);
```

(b) En Julia.

Figura 4.1. Creación de meshgrid

- Creación de precondicionadores. (ILU) En el caso de Matlab, genera la parte Lower por un lado y la Upper por el otro, sin embargo, Julia crea un objeto el cual tiene

que ser transformado en un operador para que la función BiCGStab de la librería "Krylov.jl" pueda procesarlo de forma adecuada posteriormente. (Figura 4.2)

```
[Lbicg,Ubicg] = ilu(M,struct('type','nofill'));
```

(a) En Matlab.

```
LU_ = ilu02(MPois)# ILU calculation

function forward_substitution0!(y,LU_::CuSparseMatrixCSR,v)
    ldiv!(y,UnitLowerTriangular(LU_),v)
    return y
end

function backward_substitution0!(y,LU_::CuSparseMatrixCSR,v)
    ldiv!(y,UpperTriangular(LU_),v)
    return y
end

function ldiv_ilu0!(LU_::CuSparseMatrixCSR, b, x, z)
    forward_substitution0!(z,LU_, b)        # Forward substitution with L
    backward_substitution0!(x, LU_, z)      # Backward substitution with U
    return x
end

n = length(u)
T = eltype(u)
z_ = CUDA.zeros(T, n);

opP = LinearOperator(Float64, N1*N2*(N3-2), N1*N2*(N3-2), false, false, (x, b) -> ldiv_ilu0!(LU_, b, x, z_))
```

(b) En Julia.

Figura 4.2. Definición de ILU

- Solvers de Algebra lineal (BiCGStab. Figura 4.3)

```
[p,flag] = bicgstab(M,DIV,bicgtol,bicgmaxit,Lbicg,Ubicg,pold);
```

(a) En Matlab.

```
p, history =  bicgstab(MPois,vec(DIV),vec(pold);atol=bicgtol,itmax=bicgmaxit,M=opP);
```

(b) En Julia.

Figura 4.3. Definición de BiCGStab

4.2.2. Uso de precondicionadores

Como se ha explicado en la sección 2.3.1 los precondicionadores cumplen principalmente dos funciones en el cálculo numérico: acelerar la convergencia y la velocidad de cálculo por parte de la computadora. Para analizar esto se ha realizado un "workbench" en el que se realizan una serie de análisis de velocidad y convergencia por parte del método BiCGStab (sección 2.2.2) ante diferentes tipos de aplicar el mismo precondicionador. Este precondicionador es el ILU, expuesto en la sección 2.3.1. Al momento de aplicar la factorización ILU al sistema de álgebra lineal se puede hacer de diferentes maneras, en este proyecto se han analizado 3.

- Método ILU por la derecha (RILU): En este enfoque, primero se realiza la factorización incompleta de la matriz origunal y luego se aplica el precondicionador a la matriz de coeficientes (A) y al vector derecho (b) dado el sistema lineal $Ax = b$.
- Método ILU por la izquierda (LILU): En este caso primero se transpone la matriz original (A) y luego se realiza la factorización incompleta de la matriz traspuesta. Para posteriormente aplicar este ultimo a la matriz traspuesta directamente y al vector derecho.
- Método iLU separado (SILU): Atendiendo a las siglas en inglés Separated ILU, Es una combinación de los dos anteriores, pues en lugar de aplicar la factorización incompleta solo a la matriz original o a su traspuesta, se realiza una factorización incompleta para ambas. Con esto se pretende abordar ciertas asimetrías y patrones que pueden aparecer en la matriz del sistema al aplicar ambos productos.

Es importante destacar que la elección del método depende de las características de la matriz y del solver iterativo utilizado. Es por ello que se ha decidido hacer este "workbench", para comprobar cual es el método más eficiente a nivel computacional de todos. Las dimensiones de las matrices utilizadas es de 125000x125000, lo que se corresponde con una discretización espacial 3D de 50x50x50.

Para realizar esta factorización ILU se ha utilizado una librería oficial de Julia: **IncompleteLU.jl**. Y además se han probado otros métodos de "solvers" iterativos a parte del BiCGStab, como es el caso de CGS, GMRES, DQGMRS, DIOM Y FOM, más información sobre estos métodos de resolución se pueden encontrar en la libreria **Krylov.jl** [22], que ha sido la utilizada para el desarrollo de este proyecto.

Los resultados obtenidos se resumen en la siguiente tabla:

Precondicionadores	Tiempo (s)	Iteraciones	Solver
Sin precon.	2.20	460	BiCGStab
SILU	0.95	8	BiCGStab
LILU	0.36	3	BiCGStab
RILU	1.16	8	BiCGStab
RILU	1.31	11	CGS
RILU	0.7	11	GMRES
RILU	0.66	11	DQGMRES
RILU	0.6	11	DIOM
RILU	0.7	11	FOM

Tabla 4.1. Efecto de los precondicionadores en CPU.

Se puede observar como en todos los casos el tiempo de cálculo y, sobre todo, la convergencia se reduce notablemente. El caso más eficiente es el indicado en verde en la tabla 4.1. Se puede comprobar como la velocidad se reduce un 83 % y el número de iteraciones de convergencia pasa de 460 a 3. Teniendo en consideración que para resolver el paso temporal de Runge-Kutta 4 se necesita realizar este paso 4 veces, implica que el tiempo de forma orientativa pasaría de unos 10 segundos a 2. Determinando que el mejor precondicionador y solver a utilizar es el BiCGStab con el precondicionador ILU por la izquierda.

4.2.3. Implementación cálculo GPU

El uso de cálculo GPU a un alto nivel de computación fue implementado en Julia en 2018 mediante la creación de la libreria **CUDA.jl** [15]. Esta librería creada por Tim Besard, Pieter Verstraete y Bjorn de Sutter en la universidad de Ghent, Bélgica, utiliza la metaprogramación de Julia para generar código a través de macros, de tal manera, que el lenguaje a escribir por el desarrollador sea muy simple. El uso de esta metaprogramación hace de puente entre el lenguaje de Julia de alto nivel y el lenguaje CUDA de NVidia. Este lenguaje es el que configura el hardware y en el que están escritos los kernels que se desean que se ejecuten en la GPU.

A la hora de tratar con GPUs el funcionamiento suele ser de la siguente manera.

- Definir las operaciones que se desean realizar.
- Reservar la memoria deseada con las variables con las que se van a realizar los cálculos.

Los dos puntos importantes del código donde este tipo de cálculo es de suma importancia son el cálculo del sistema algebraico lineal de la ecuación de Poisson y el cálculo de las matrices de derivadas.

Cálculo ecuación Poisson

El cálculo de BiCGStab definido por la librería **Krylov.jl** permite el cálculo en GPU. Por lo que el punto clave en el cálculo en GPUs NVIDIA en Julia es el paso de las variables calculadas en el setup. En este punto puede aparecer la duda de porqué todo el programa no se ha realizado en el coprocesador. La razón por la que las variables no pueden definirse en primera instancia en la GPU es que a la hora de crear las matrices de diferencias finitas, se realizan con referencias escalar, es decir, se rellena posición por posición.

En el caso de este proyecto se van a utilizar 2 tipos diferentes de arreglos. Por un lado se utilizarán para los Arrays, CuArrays y para SparseMatrixCSC (Matrices dispersas guardadas por columnas), CuSparseMatrixCSR (Matrices dispersas guardadas por filas). Esta operación puede contemplarse en la imagen 4.4.

```
MPois = CuSparseMatrixCSR(MPois)
u = CuArray(u)
```

Figura 4.4. Implementación de la variables a la GPU

De esta manera, para analizar la mejora de la velocidad mediante el uso de GPU se ha creado un "workbench" idéntico al anterior. De esta manera se observará como se comportan los diferentes tipos de precondicionadores y métodos de resolución iterativa algebraica.

Precondicionadores	Tiempo (s)	Iteraciones	Solver
Sin precon.	0.148	430	BiCGStab
LILU	0.017	2	BiCGStab
RILU	0.296	82	BiCGStab
LILU	0.157	526	CGS
LILU	0.010	4	GMRES
LILU	0.010	4	DQGMRES
LILU	0.009	4	DIOM
LILU	0.010	4	FOM

Tabla 4.2. Efecto de los precondicionadores en GPU.

Se obseva como al realizar las operaciones en la GPU, estos se aceleran sin precondicionadores un 93 %, si se compara con el método más rápido el caso sin precondicionador y en CPU un 95 %. Esto implica que se pasa de que una iteración temporal pase de tardar 10 segundos a medio segundo. Se han destacado dos porque en cuanto a velocidad ambos son muy eficientes, sin embargo parece en un principio que el método DIOM es bastante

más rápido que el BiCGStab. Esto acaba siendo erroneo a la hora de llamarlo de forma iterativa se ha comprobado que la diferencia no es tan notable para el tamaño del problema deseado. También llama la atención como en la CPU los métodos más rápidos son los del precondicionamiento por la derecha y en la GPU por la izquierda. Esto se debe a los tipos de matrices dispersas que se utilizan. En el caso de la CPU se han guardado por filas, es por ello que el producto por la derecha sea más rápido que por la izquierda, mientras que la GPU lo guarda por columnas.

Cálculo de las matrices de derivadas

A la hora de calcular las matrices esparcidas derivadas según las ecuaciones en la sección 2.2.1, en el código original de Matlab se calcular la derivada término a término en toda la matriz esparcida, de tal forma que se usa indexación escalar. Además, el código original utiliza numerosos comandos condicionales en función de cada posición a calcular. Este algoritmo hace que el cálculo sea muy lento, pues solo se puede realizar de forma secuencial. Por esto es que para acelerar este cálculo se ha tenido que modificar la estructura del código, no sólo vectorizando los cálculos si no los operandos condicionales.

El código original de Matlab se compone de tres bucles for, que recorren las tres dimensiones de cada matriz y si cumple ciertos requerimientos, se calcula la derivada y se añade a la matriz esparcida. Si se quiere vectorizar este cálculo, es decir, realizar todos los cálculos simultaneamente, es necesario que los valores FZ0, FZA y FZG sean vectores de tal forma que los cálculos sean de arreglos matemáticos en vez de individuales.

```
1 for k = inz'-1
2     for j = inx'
3         for i = iny'
4             FZ0 = FZ(i,j,k+1);
5             FZA = FZ(i,j,air(k));
6             FZG = FZ(i,j,ground(k));
```

Listing 4.1: Inicialización de las variables en Matlab

```
    FZ0 .= FZ[:, :, 2:end-1];
    FZA .= FZ[:, :, air] ;
    FZG .= FZ[:, :, ground];
```

Listing 4.2: Inicialización de las variables en Julia

Tras ello se procede a crear la estructura booleana de los condicionales. En el ejemplo expuesto se pretende rellenar la posición con un valor si no se halla en la pared y con otra si se encuentra en la pared. Para hacer esto de forma vectorial, se procede de la misma manera de antes, solo que sí cumple la condición se rellena con un 1 y si no, con un 0.

```
                if AG(A0(i,j,k)) >= 1
                    ...
                else
                    ...
                end
```

Listing 4.3: Condicionales en Matlab

```
    NO_GROUND = A0_AG_1D .>= 1; #First condition of if
    GROUND = A0_AG_1D .< 1; #else
```

Listing 4.4: Condicionales en Julia

A continuación, en Matlab se introduce directamente el valor en la posición deseada, sin embargo en Julia el proceso es un poco más complejo, pues hay que calcular la matriz con todos los valores, pasar la información a la GPU, aplicar el condicional a través de su arreglo, tanto valores como posiciones, deshacerse de los ceros y añadir los valores a sus posiciones. Este proceso más largo se debe a que no es posible realizar esta operación directamente en una GPU debido a su naturaleza multithreading.

```
    M(A0(i,j,k),AG(A0(i,j,k))) = -1/FZ0 * FZ0/(FZ0+FZG);
```

Listing 4.5: Condicionales en Matlab

```
    #value calculation on GPU
    @. P1_GPU = (-1/FZ0_1D_GPU*
```

```
3     FZ0_1D_GPU/(FZ0_1D_GPU+FZG_1D_GPU));
4
5     # Get data into CPU
6     P1 = Array(P1_GPU);
7
8     # Get data of cells that are not on the ground
9     P1 = P1 .* NO_GROUND;                    # values            CPU
10    A0_AG_1D_NO_G = A0_AG_1D .* NO_GROUND;   # M positions       CPU
11
12    P1_miss = filter(x -> x !=0, P1);                          #NO_Ground
    Cells values dismiss    CPU
13    A0_AG_1D_miss = filter( x-> x != 0, A0_AG_1D_NO_G);        #NO_Ground
    Cells values dismiss    CPU
14
15    M[A0_AG_1D_miss] = P1_miss; #Transfer data to M sparse matrix
                          CPU
```

Listing 4.6: Julia diff code scheme

Se puede observar como el código se complica y se alarga bastante, pues requiere de más pasos para tratar los datos de forma vectorizada. A continuación se va a exponer cuánto se ha acelerado este procedimiento respecto al código original de Matlab. Para ello se van a analizar 3 casos: Código original de Matlab, solver de Julia en CPU y solver de Julia en GPU.

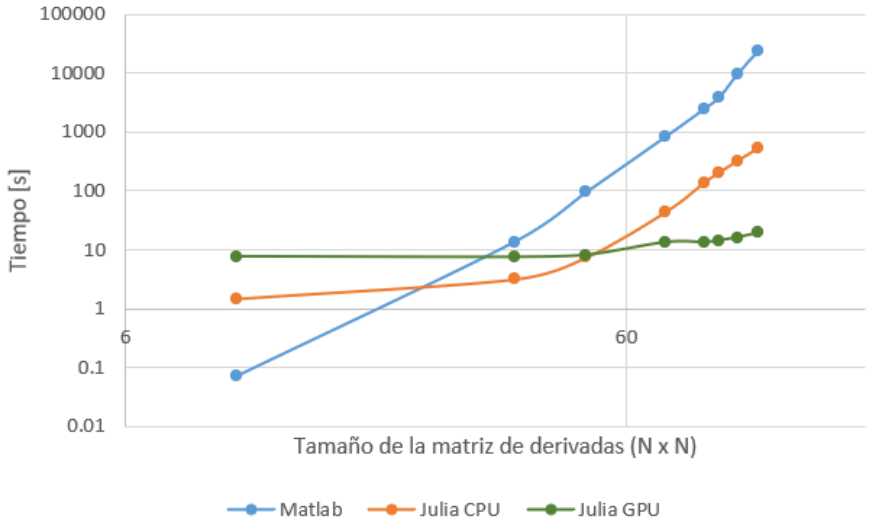


Figura 4.5. Velocidad de cálculo de todas las matrices de derivadas frente al tamaño de las mismas

En la imagen se puede comprobar como Julia es, de hecho, un lenguaje mucho más rápido que Matlab para el trato de operaciones básicas, utilizando indexación escalar. Para tamaños de matrices pequeños la CPU se comporta de forma más rápida que la GPU, esto se debe a que el paso de información entre GPU y CPU tiene un coste elevado en relación con tamaños pequeños de matrices, sin embargo, cuando el tiempo de computación empieza a ser relevante, el cálculo GPU es cuando empieza a destacar más respecto a la CPU.

De esta manera, para el caso de un tamaño de 92x92 el tiempo que tarda la CPU, en Matlab y Julia respectivamente es de unos 3855 y 203 segundos, mientras que Julia con cálculo en GPU tarda solo unos 14 segundos. El cálculo para dimensiones mayores se hace inviable para el caso de Matlab y Julia en CPU. Sin embargo el orden de magnitud se mantiene constante Para la GPU, pues lo único que hace es almacenar mayor memoria y definir más hilos para realizar operaciones de forma simultánea.

Estudio de la memoria ocupada

Puesto que el límite de operabilidad de este problema reside en la memoria, se ha realizado un estudio de cual sería el tamaño máximo que se puede utilizar para realizar los cálculos. En un principio, en la GPU se utilizan 2 tipos diferentes de tipos: Matrices dispersas y Matrices densas.

Existen diferentes métodos para guardar la información de las matrices dispersas, en este caso se ha utilizado la forma de serie de Julia, que el método CSC. Este método consiste en guardar la información por columnas. Por ello la memoria de la matriz se estructura en tres vectores, uno el cual guarda las columnas en los que hay valores, otro el cual guarda las filas en aquellos en los que hay valores y por último el que guarda el valor en la ubicación que indican los otros dos vectores. Por un lado los dos primeros son enteros simples, por lo que cada índice tiene un coste de memoria de 4 bytes mientras que el último es un tipo flotante de doble precisión de 8 bytes. Además, tiene un par de métodos más los cuales guarda información básica como las dimensiones básicas ocupando un total de 20 bytes.

La memoria que ocupa una matriz densa es simple de calcular, pues es básicamente las dimensiones totales de la matriz multiplicado por 8 bytes que ocupa cada número de tipo doble.

Aunque a estas matrices habría que añadirles cierta memoria que se ocupa antes de iniciar los cálculos utilizada para calcular las matrices de derivadas y para definir las funciones que van a ser utilizadas por el solver posteriormente, para este análisis se ha considerado despreciable.

Este estudio queda recogido en la siguiente tabla, donde en la última columna está escrita la fórmula para calcular la memoria ocupada por las matrices del problema.

Tipo	Cantidad	Tamaño	Memoria (Bytes)
Matriz dispersa	11	$N \times N \times N$	$N^3 \cdot 2 \cdot 14$
Matriz densa	9	$N \times N \times N$	$N^3 \cdot 8$
Matriz densa	3	$N \times N \times (N+2)$	$N^2 \cdot (N+2) \cdot 8$

Tabla 4.3. Tabla de datos utilizados en la GPU.

Asumiendo que la GPU utilizada, NVIDIA RTX A5000, tiene una memoria total de 24Gb, de la cual se puede utilizar el 80 % para que la GPU pueda realizar operaciones internas sin que colapse, se puede ocupar un total de 19Gb, despejando del sistema anterior expuesto en la tabla 4.3 se obtiene que el tamaño máximo recomendable para una operabilidad segura es de $N = 365$ o un total de $48 \cdot 10^6$ de nodos, si se quiere utilizar diferentes resoluciones en cada dirección.

Capítulo 5

Validación y Resultados

Para la validación se va utilizar el código que simula un caso turbulento con un $Re_\tau = 180$ y se va a comparar con la referencia de datos del estudio realizado por Moser [23]. En todo solver turbulento la inicialización es un punto clave, pues debido a su naturaleza la disipación de la energía del flujo puede acabar dando lugar a la desaparición de la turbulencia. Así se obtendría un flujo laminar. Esto ha complicado la correcta convergencia del programa en Julia. Se han realizado dos análisis: uno partiendo de las condiciones iniciales de Matlab y otro replicando el programa de Matlab al completo.

En el código orignal de Matlab la obtención de las condiciones iniciales se hace a una resolución de 36x36x36 y consiste en dos pasos, por un lado se calcula el campo fluido laminar con unas 50 pasos temporales. Después se añade unas fluctuaciones random, siguiendo una distribución normal, al campo de velocidades de tal forma que se fuerza la existencia de flujo turbulento. Y tras realizar unas 1000 iteraciones, se interpola el campo de velocidades para pasar de una dimensión 36 a 72, de esta forma las escalas más pequeñas de la turbulencia son despreciadas y es más difícil que se disipe la energía y el problema deje de ser turbulento.

Partiendo de este campo de velocidades inicial obtenido en Matlab se han obtenido los

siguientes resultados. La figura 5.1 muestra el campo fluido completamente desarrollado. Se puede observar como en las paredes superior e inferior la capa límite turbulenta está desarrollada y que la velocidad se aproxima a cero conforme se acerca a estas. En la pared superior se observa un corte en el plano x-y en $y^+ \approx 10$. Donde se pueden observar las vetas o líneas turbulentas dentro de la capa límite.

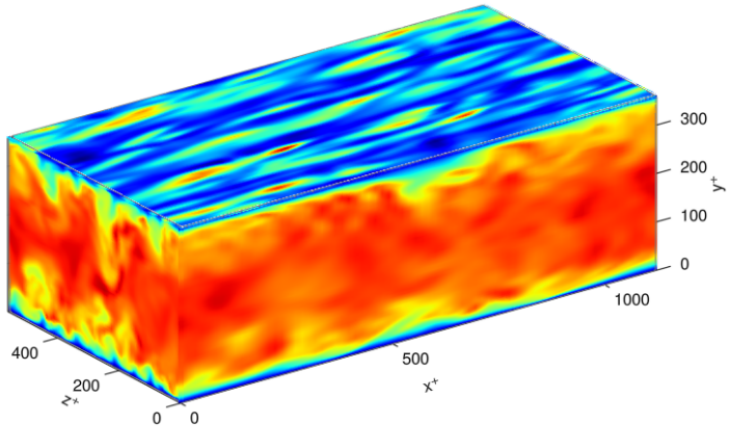


Figura 5.1. Velocidad U^+ del flujo desarrollado

El carácter turbulento del flujo se representa en la figura 5.2 mostrando el perfil de la velocidad media entre las paredes del canal junto con ejemplos locales en diferentes secciones de este. De esta manera se puede observar como el flujo turbulento fluctúa entorno a la media. Las velocidades de fluctuación (u^+, v^+, w^+) y la velocidad media (U^+) han sido normalizadas con la velocidad de fricción de tal manera que $u^+ = u/u_\tau$ y la escala de longitud viscosa de tal forma que $y^+ = yu_\tau/\nu$. En este paso se ha utilizado la definición de $u_\tau = \sqrt{\nu \frac{du}{dy}}$, donde $\frac{du}{dy}$ es el gradiente de la velocidad en la dirección normal a la pared evaluada numéricamente a partir de la velocidad media en la dirección principal del flujo.

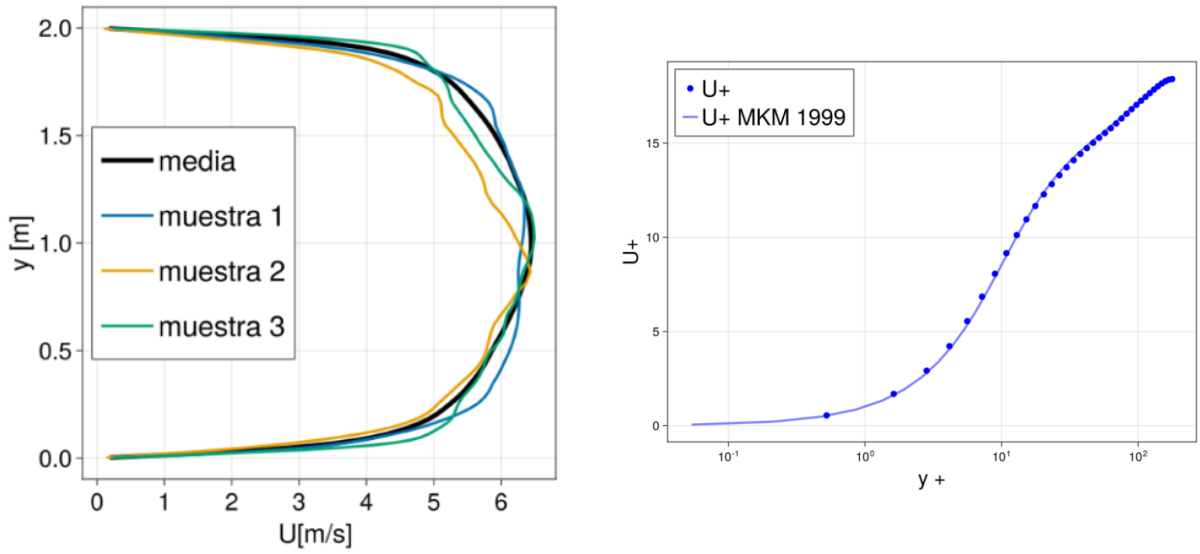


Figura 5.2. Velocidad media de la corriente en unidades físicas a través del canal junto con perfiles instantáneos de varios puntos de muestreo (izq.). Velocidad media a lo largo de la corriente en unidades de pared en escala log-lin (dch.)

Tanto en la figura 5.2, en el panel derecho, como en la 5.3 se puede comprobar como las velocidades promediadas en la dirección de la corriente y la desviación estándar promediada en tiempo y espacio de las componentes de la velocidad concuerdan con los datos de referencia calculadas por Moser. De hecho, esto no es en absoluto una sorpresa ya que la resolución numérica es muy cercana a la resolución DNS típica para este caso particular como la utilizada en estudios previos [19] [23]. Además en la figura 5.3 se compara con los datos obtenidos por el solver de Matlab y se observa claramente como las tendencias son las mismas en ambas soluciones.

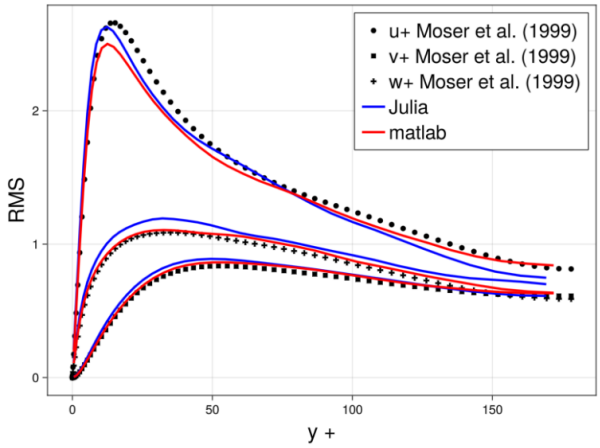


Figura 5.3. Fluctuación media de la velocidad a través del canal

Cabe mencionar como en el panel derecho de la figura 5.2 se puede observar las diferentes regiones de la capa límite viscosa. La cual se divide en tres partes, la subcapa viscosa ($Y^+ < 5$), donde la viscosidad domina al fluido, la capa intermedia, donde se da la transición entre la región donde la viscosidad es dominante y la parte turbulenta y el área logarítmica ($y^+ > 30$) a partir de la cual los esfuerzos turbulentos dominan el fluido.

Una vez concluido que la implementación del código a Julia ha sido correcta y que se ha logrado uno de los objetivos que se perseguían. En la tabla 5.1 se expone la diferencia de tiempos entre Julia y Matlab para este caso en concreto en el que las condiciones iniciales son las mismas.

Time (s)	Matlab	Julia
Precon	960	11
Solver	1612	813
Total	2572	824

Tabla 5.1. Tabla que compara tiempos de ejecución entre Matlab y Julia

Por otro lado se ha intentado reproducir el caso completo de Matlab, el cual para conseguir las condiciones iniciales requería de correr el código 2 veces, en una única ejecución en Julia.

Sin embargo, al ejecutarlo replicando a Matlab, no se obtiene la convergencia deseada en 1000 iteraciones. Esto se muestra en la imagen 5.4 Por ello, para obtener unas condiciones iniciales mejores se decidió aumentar el número de iteraciones en el segundo paso pasando de 1000 a 1500. De esta manera, con un coste de tiempo mínimo se obtiene una solución acorde a la de Moser y Matlab.

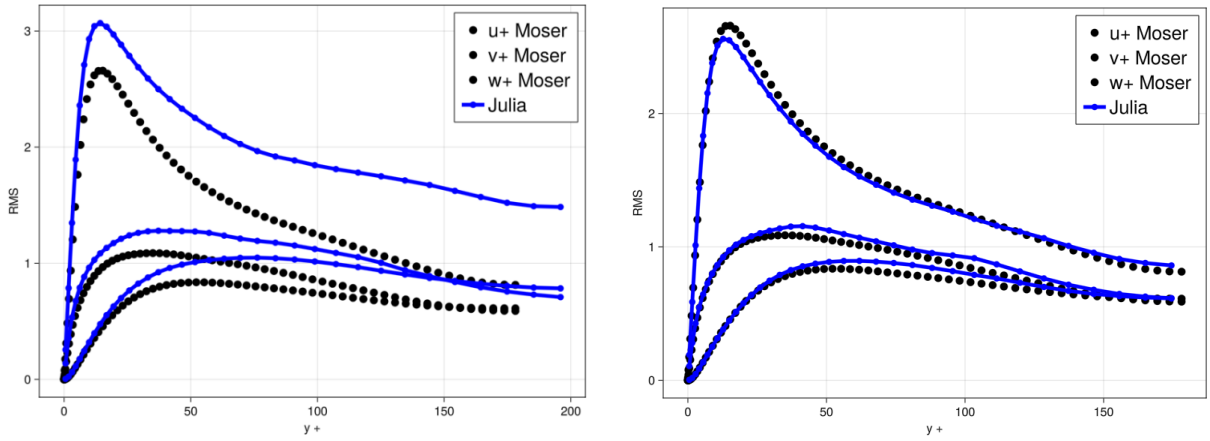


Figura 5.4. Fluctuación media de la velocidad a través del canal replicando Matlab y con la nueva configuración

El tiempo que se tarda en ejecutar esta solución es de 1457 segundos en total, mientras que en Matlab tardaría 2572 segundos a lo que habría que sumar las 2 ejecuciones para obtener las condiciones iniciales.

Capítulo 6

Conclusiones

El objetivo inicial de este trabajo final de máster ha sido la aceleración mediante tres procesos diferentes la ejecución de un solver numérico DNS desarrollado inicialmente en Matlab.

Por un lado se ha realizado la migración de código de Matlab a Julia, un lenguaje de programación relativamente nuevo pensado para cálculos de alto rendimiento. Se trata de lenguaje de código libre, lo que al contrario que Matlab, permite usarlo de forma gratuita. En primera instancia se demostró una mayor velocidad de computacional sobre todo para cálculos indexados escalarmente, hasta un **95 %** más rápido. No obstante, a la hora de utilizar funciones de bibliotecas del propio lenguaje se queda notablemente atrasada respecto a Matlab, siendo hasta un **60 %** más lento.

Posteriormente aplicando precondicionadores se consiguió acelerar el cálculo del sistema pasando de 2.2 segundos a 0.36 segundos, reduciendo el tiempo de cálculo un **84 %**.

Por último se implementó el uso de coprocesadores GPU. Con el que se logró acelerar el cálculo del sistema pasando de 0.36 segundos a 0.017, obteniendo una mejora del **95 %**. También se analizó el uso de otros solvers algebraicos a parte del BiCGStab, pero el

rendimiento era similar a este. Esta tecnología no se ha utilizado solo para acelerar el solver, sino también para acelerar los cálculos de las matrices de derivadas que se utilizan dentro del cálculo. Solo con la implementación a Julia este proceso ya mejoró por encima del 95 % a la hora de calcular matrices mayores de dimensión 50. Utilizando la aceleración GPU, con un coste de longitud y complejidad de código, estos tiempos se logran reducir otro 95 %.

Para obtener una conclusión más apreciable se compararon los tiempos totales al correr el mismo código en Matlab y en Julia con matrices de dimensión 72, divididos en dos partes. La de preprocesado y la resolución del mismo. En el preprocesado, se han obtenido unos tiempos de 960 y 11 segundos respectivamente. En el caso del solver se han obtenido unos resultados de 1612 y 813 segundos respectivamente.

Dado estos resultados, se puede concluir que la aceleración del código ha sido muy notable. No obstante la convergencia del solver el Julia se ha visto afectada debido al cálculo de las condiciones iniciales, haciendo que requiera más iteraciones para converger a la solución. Por otro lado, si se desea acelerar más el código, uno de los métodos más efectivos sería adquirir una GPU más apta para este tipo de cálculos de doble precisión, pues la utilizada esta más enfocada en cálculo de precisión simple, como el que utilizan las Inteligencias Artificiales.

Precisamente las futuras posibles lineas de desarrollo pueden encontrarse en este punto. Sería interesante pues, analizar las diferencias estadísticas entre las soluciones de precisión simple y doble con el fin de analizar la viabilidad de utilizar la simple y de esta manera acelerar aún más el código. Además de esto, sería interesante invertir tiempo en encontrar una forma más optimizada de obtener unas condiciones iniciales que reduzca el tiempo de ejecución y mejorar el postprocesado de los datos calculados por el solver.

Bibliografía

[1] A. M. TURING. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 10 1950.

[2] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2006.

[3] Claudio Canuto, M Yousuff Hussaini, Alfio Quarteroni, and Thomas A Zang. *Spectral methods: fundamentals in single domains*. Springer Science & Business Media, 2007.

[4] Claudio Canuto, M Yousuff Hussaini, Alfio Quarteroni, and Thomas A Zang. *Spectral methods: evolution to complex geometries and applications to fluid dynamics*. Springer Science & Business Media, 2007.

[5] V. Vuorinen, M. Larmi, P. Schlatter, L. Fuchs, and B.J. Boersma. A low-dissipative, scale-selective discretization scheme for the navier–stokes equations. *Computers & Fluids*, 70:195–205, 2012.

[6] Alexandre Joel Chorin. A numerical method for solving incompressible viscous flow problems. *Journal of computational physics*, 135(2):118–125, 1997.

[7] Lloyd N Trefethen. *Spectral methods in MATLAB*. SIAM, 2000.

[8] V. Vuorinen and K. Keskinen. Dnslab: A gateway to turbulent flow simulation in matlab. *Computer Physics Communications*, 203:278–289, 2016.

[9] Charles Hirsch. *Numerical computation of internal and external flows: The fundamentals of computational fluid dynamics*. Elsevier, 2007.

[10] Joe D Hoffman and Steven Frankel. *Numerical methods for engineers and scientists*. CRC press, 2018.

[11] Benjamin Schmiedel. Runge-kutta 4th order scheme.

[12] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[13] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[14] William F Godoy, Pedro Valero-Lara, T Elise Dettling, Christian Trefftz, Ian Jorquera, Thomas Sheehy, Ross G Miller, Marc Gonzalez-Tallada, Jeffrey S Vetter, and Valentin Churavy. Evaluating performance and portability of high-level programming models: Julia, python/numba, and kokkos on exascale nodes. *arXiv preprint arXiv:2303.06195*, 2023.

[15] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[16] GoogleFinance. Nvidia stock through the years.

[17] Pradeep Gupta. Nvidia technical blog. cuda refresher: The cuda programming model.

[18] NVIDIA. Nvidia rtx a5500 datasheet.

[19] V. Vuorinen, J.-P. Keskinen, C. Duwig, and B.J. Boersma. On the implementation of low-dissipative runge–kutta projection methods for time dependent flows using openfoam®. *Computers & Fluids*, 93:153–163, 2014.

[20] Simon Danisch and Julius Krumbiegel. Makie.jl: Flexible high-performance data visualization for Julia. *Journal of Open Source Software*, 6(65):3349, 2021.

[21] D. Orban, A. S. Siqueira, and contributors. LinearOperators.jl. `https://github.com/JuliaSmoothOptimizers/LinearOperators.jl`, September 2020.

[22] A. Montoison, D. Orban, and contributors. Krylov.jl: A Julia basket of hand-picked Krylov methods. https://github.com/JuliaSmoothOptimizers/Krylov.jl, June 2020.

[23] Robert D Moser, John Kim, and Nagi N Mansour. Direct numerical simulation of turbulent channel flow up to re τ= 590. *Physics of fluids*, 11(4):943–945, 1999.

Appendices

Apéndice A

Pliego de condiciones

En el presente apartado se expone la normativa vigente a cumplir en el desarrollo del proyecto de fin de máster sobre seguridad, salud e higiene. Se recurre al Real Decreto 488/1997 del 14 de abril y el Real Decreto 486/1997.

A.1. Real Decreto 488/1997 de 14 de abril

El presente decreto establece las disposiciones mínimas de seguridad y salud para el empleo por parte de los trabajadores de equipos que incluyan pantallas de visualización.

- Equipo. El empleo del equipo disponible no debe ser una fuente de riesgo para los trabajadores, por lo que se deben tener en cuenta los siguientes aspectos:
  - Pantallas.
    - Se deben emplear pantallas con caracteres bien definidos y configurados de forma precisa con dimensiones suficientes y espaciado adecuado.
    - La imagen proporcionada por las pantallas debe ser estable y sin destellos

    - La luminosidad, contraste de caracteres y fondo de la pantalla deben de ser fácilmente ajustables al entorno de trabajo, además de ser orientable a las necesidades del trabajador.
    - Posibilidad de empleo de un pedestal independiente o mesa regulable para la pantalla.
  - Teclado.
    - Teclado inclinable e independiente de la pantalla que permita adoptar al trabajador una postura cómoda.
    - Debe ser mate para evitar los reflejos y sus teclas deben ser legibles desde la posición normal de trabajo
  - Mesa o superficie de trabajo.
    - Mesa poco reflectante con dimensiones suficientes que permita la colocación óptima de la pantalla, teclado y documentos necesarios para desempeñar la labor del trabajador.
    - El espaciado debe permitir a los trabajadores la adopción de una posición cómoda.
  - Asiento de trabajo.
    - El asiento debe ser estable y proporcionar una postura confortable, con altura regulable y reclinable.

- Entorno
  - Espacio
    - El puesto de trabajo debe tener una dimensión suficiente para permitir cambios y movimientos del trabajador.
  - Iluminación
    - La iluminación debe garantizar unos niveles adecuados entre la pantalla y el entorno, evitando deslumbramientos y reflejos en la pantalla u otras partes del equipo.

- Reflejos y deslumbramientos.
  - Los puestos de trabajo deben disponerse de forma que las fuentes de luz como pueden ser ventanas y otras aberturas no provoquen reflejos y deslumbramientos molestos de la pantalla de trabajo.
  - Se debe equipar las ventanas con dispositivos de cobertura adecuados.
- Ruido El ruido no debe perturbar en ningún caso la atención de los trabajadores, así como el calor procedente de los equipos no debe ocasionar molesta en los trabajadores.
- Emisiones Las emisiones deben reducirse a niveles bajos desde el punto de vista de la seguridad y la salud de los trabajadores
- Humedad Debe crearse y mantenerse una humedad aceptable.

- Interconexión ordenador/persona
  - El programa a emplear debe estar adaptado a la tarea que debar realizarse, debe ser fácil de utilizar y adaptarse al nivel de conocimientos y experiencia por parte del trabajador.
  - Los sistemas deberán proporcionar informaciones a los trabajadores sobre su desarrollo y mostrar la información en un formato adecuado de los operadores.

A.2. Real Decreto 486/1997

El presente Real Decreto establece las disposiciones mínimas de seguridad y de salud que deben ser aplicadas a los lugares de trabajo, este cuenta con las siguientes secciones importantes.

A.2.1. Condiciones generales de seguridad en los lugares de trabajo

- Seguridad estructural Los edificios de trabajo deben poseer una estructura apropiada a su desempeño, deben tener solidez y resistencia para soportar cargas y disponer de un sistema armado que asegure su estabilidad.
- Espacio de trabajo y zonas de trabajo
  - Las dimensiones de los edificios de trabajo deben permitir que los trabajadores realicen sus funciones sin riesgos para su seguridad y salud.
  - Deben tomarse las medidas apropiadas para la protección del trabajador al acceder a zonas de los lugares de trabajo donde su integridad pueda verse afectada por riesgo de caída de objetos o exposición a elementos agresivos, estando estas claramente señalizadas y disponiendo de un sistema que no permita acceder a personal no autorizado a esas zonas
- Suelos, aberturas, desniveles y barandillas
  - Los suelos deben ser fijos y estables.
  - Las aberturas o desniveles deben ser protegidas mediante barandillas u otros sistemas de protección, las barandillas deben ser de materiales y altura adecuada.
- Tabiques, ventanas y vanos
  - Los tabiques deben estar claramente señalizados y fabricados de materiales seguros evitando que los trabajadores puedan golpearse con los mismos.
  - Se debe poder realizar de forma segura las operaciones de abertura, ajuste o fijación de ventanas, pudiendo ser limpiadas sin riesgo para los trabajadores que realicen esta tarea.
- Vias de circulación

- Las vías de circulación deben poder emplearse de forma sencilla y con seguridad para los trabajadores que circules por ellas, la anchura de las vías debe permitir el uso simultáneo de medios de transporte y peatones con una separación de seguridad suficiente.
- El trazado de las vías de circulación deberá estar siempre claramente señalizado

- Puertas y portones
  - Las puertas transparentes deben tener una señalización a la altura de la línea de visión, este tipo de superficies deben protegerse de la rotura cuando pueda suponer un problema para los trabajadores.
  - Las puertas correderas deben ir provistas de un sistema de seguridad
  - Las puertas y portones de accionamiento mecánico deben funcionar sin riesgo para los trabajadores, contando con dispositivos de parada en el caso de que sea necesario.
  - Los portones destinados a la circulación de vehículos deben poder ser utilizados por los peatones con seguridad.

- Rampas, escaleras fijas y de servicio
  - Los pavimentos de las rampas, escaleras y plataformas de trabajo deben ser de material no resbaladizo
  - Los peldaños de las escaleras deben tener las mismas dimensiones, prohibiendo el uso de escaleras de caracol, en el caso de trabajar con escaleras mecánicas, deberán garantizar la seguridad de los trabajadores que las utilicen

- Vías y salidas de evacuación
  - Las vías y salidas de evacuación deben desembocar lo más directamente posible al exterior o una zona de seguridad, permitiendo a los trabajadores evacuar los lugares de trabajo rápidamente, estas deben estar señalizadas y no obstruidas por ningún objeto

  - Las vías y salidas de evacuación deben desembocar lo más directamente posible al exterior o una zona de seguridad, permitiendo a los trabajadores evacuar los lugares de trabajo rápidamente, estas deben estar señalizadas y no obstruidas por ningún objeto
  - Las puertas de emergencia deberán abrirse hacia el exterior y no estar cerradas, todas ellas señalizadas de manera adecuada para poder ser empleadas sin ayuda

- Condiciones de protección contra incendios Los lugares de trabajo deberán estar equipados con dispositivos adecuados para la extinción de incendios y detectores en el caso de que fuesen necesarios.
- Instalación eléctrica Las instalaciones eléctricas no deben entrañar riesgos de incendio o explosión, asegurando la protección de los trabajadores.

A.2.2. Orden, limpieza y mantenimiento

Los lugares de trabajo deben ser limpiados periódicamente y mantenerlos en condiciones higiénicas adecuadas. Las operaciones de limpieza no deben constituir por si mismas una fuente de riesgo para los trabajadores que desempeñen esta tarea.

A.2.3. Condiciones ambientales de los lugares de trabajo

La exposición al as condiciones ambientales de los lugares de trabajo no deben suponer un riesgo para los trabajadores ni suponer para ellos una fuente de incomodidad, por lo tanto, deberán evitarse temperaturas y húmedas altas, cambios bruscos de temperatura, olores desagradables y corrientes de aire molestas entre otros. El asilamiento térmico del lugar de trabajo debe adecuarse a las condiciones climáticas de la zona.

A.2.4. Iluminación de los lugares de trabajo

La iluminación debe ajustarse a las características de la actividad que se efectúe en el lugar de trabajo, siempre que sea posible, los lugares de trabajo tendrán iluminación natural complementada con iluminación artificial, buscando una distribución de niveles de iluminación lo más uniforme posibles, evitando deslumbramientos directos que pueda producir la luz solar o artificial. Un fallo en el sistema de iluminación no debe suponer un riesgo en la seguridad de los trabajadores ni originar riesgos eléctricos, de incendio o de explosión.

A.2.5. Servicios higiénicos y locales de descanso

En los lugares de trabajo será necesario disponer de:

- Agua potable.
- Vestuarios, duchas, lavabos y retretes.
- Locales de descanso.

A.2.6. Servicios higiénicos y locales de descanso

Los lugares de trabajo deben disponer de material de primeros auxilios en el caso de accidentes y con facilidad para acceder al mismo; en el caso de que el lugar de trabajo cuente con más de 50 trabajadores, se debe disponer de un lugar destinado a primeros auxilios y atención sanitaria claramente señalizados.

Apéndice B

Presupuesto

B.1. Coste de personal

Este coste incluye a los trabajadores que han formado parte del proyecto. Estos son el alumno, el tutor del laboratorio de este proyecto y el tutor en la universidad. El sueldo correspondiente al alumno se considera uno típico de alumno de prácticas en un laboratorio. El coste total y las horas realizadas se muestran a continuación:

	Horas	Sueldo (€/h)	Subtotal (€)
Tutor Universidad origen	2	20	40
Tutor Universidad destino	15	20	300
Alumno	675	6	4050
Total			4390

B.2. Coste de software

Para el desarrollo de este trabajo final de master se ha trabajado en linux, por lo que su licencia es completamente gratuita. En cuanto al entornos de programación se han utilizado Visual code, que también es open source y Matlab, el cual tiene un coste individual que roza los 2000€/año. Además todas las librerias utilizadas de Julia son también free source. Por lo que se puede extraer la siguiente tabla:

	Licencia(€/año)	Tiempo	Subtotal (€)
Matlab	2000	6 meses	1000
Visual Studio	-	-	0
Linux	-	-	0
Julia	-	-	0
Total			1000

B.3. Coste de hardware

Esta sección es la que más dinero se ha invertido, pues para el correcto desarrollo del trabajo se adquirió una GPU Nvidia RTX A5000 y una nueva unidad de potencia para el ordenador para poder utilizar dos GPUs simultaneamente. Además de esto se supondrá un gasto teórico de 9 céntimos la hora de obsolescencia del ordenador.

	Coste(€/h)	Tiempo	Subtotal (€)
GPU	1840	-	1840
Unidad Potencia	193	-	193
Ordenador	0.09	675	60.7
Total			2093.7

B.4. Presupuesto total

Considerando el desglose del presupuesto realizado, sumando los subtotales y aplicando el impuesto de valor añadido reglamentario del 21 %, se obtiene un presupuesto final del proyecto de **nueve mil cincuenta y cinco euros**

Personal		4390
Software		1000
Hardware		2093.7
0.09	subtotal	7483.7
	IVA 21 %	1571.43
Total		9055