UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO

Facultad de Ingeniería Mecánica

TESIS

"IMPLEMENTACIÓN DEL CÁLCULO DE ALTO RENDIMIENTO EN OPENFOAM PARA EL ANÁLISIS COMPUTACIONAL EN EL TUBO DE ASPIRACIÓN DE LA TURBINA T-99"

QUE PARA OBTENER EL GRADO DE MAESTRO EN CIENCIAS EN INGENIERÍA MECÁNICA

PRESENTA:

Mario Ernesto Jiménez Rosales

ASESORES:

Dr. en Ciencias en Ingeniería Mecánica Sergio Ricardo Galván González

Dr. en Energías Renovables y Eficiencia Energética Carlos Rubio Maya

MORELIA, MICH. ENERO 2014

Gracias a Dios, aquellos que amo, aquellos que me apoyaron profesionalmente y aquellos que me han regalado una sonrisa incondicional.

Foto: Mi querida madre con nuestra mascota.

RESUMEN

La energía hidroeléctrica se genera de manera limpia a baja entropía, sin embargo la mayoría de las plantas que generan la electricidad son antiguas y necesitan rehabilitación. Para lograrlo, una vía es el estudio del flujo a través de la turbina. Las principales características del flujo en las turbinas hidráulicas pueden ser resueltas por medio de modelados numéricos realizados con la Dinámica de Fluidos Computacional (*CFD*). Actualmente, una simulación lo mas aproximada a lo real es computacionalmente muy costosa. Por lo tanto, para la aplicación de modelos de turbulencia y esquemas de discretización que demandan más memoria de cálculo, se necesita de computadoras o configuraciones más potentes para lograr resultados confiables y detallados.

Las tendencias actuales en el cálculo de alto rendimiento o *High Performance Computing* (*HPC*), es su aceleración por medio de Unidades Procesadoras de Gráficos (*GPU*'s) a través del paralelismo fino de aplicaciones. Algunos algoritmos en *CFD* contienen la cualidad de la aceleración por medio de *GPU*. Este es el caso del código *CFD* con licencia *GPL* o bien de distribución gratuita, OpenFOAM, el cual tiene la ventaja de tener el código abierto (sin limitaciones en la cantidad de nodos que se pueden usar y el acceso a los algoritmos de los cálculos matriciales).

Esta tesis presenta el análisis del paralelismo de las ecuaciones de Navier-Stokes 3D utilizando los *GPU*'s del flujo incompresible en el código *CFD*; OpenFOAM, con aplicación al tubo de aspiración de la turbina hidráulica T-99. Con los resultados de este trabajo se espera una visualización completa del flujo y una reducción importante del tiempo de cálculo por medio de una comparación cuantitativa, para este importante dispositivo de la turbina.

Palabras clave: HPC, CFD, GPU, OpenFOAM, T-99.

ABSTRACT

Hydroelectricity is a form of energy obtained via a low-entropy producing process carried out in hydropower stations. Most of power plants are old and in need of refurbishment. The development of the numerical methods and computer power during the last decades has made the Computational Fluids Dynamics (CFD) an attractive tool design for this purpose. The general features of the flow in water turbines can be resolved, but in order to study the flow in detail, large HPC (High Performance Computing) facilities are required as high turbulence models, discretization schemes and grid density are needed.

Actual tendencies in HPC using Graphics Processor Units (GPU) that are designed to do graphics rendering that have emerged as massively-parallel "co-processors" for the central processing unit (CPU). This work presents the parallel implementation of the Navier-Stokes 3D with application to the T-99 draft tube, benchmark for speed up the calculus in different cases are presented. Some parameters existing in information technology have been used to measure the numerical and computing performance of the simulations. We use Free Source Software such as GNU/linux as Operative System and OpenFoam as Fluid Simulator. GPU is expected to offer affordable low cost, high performance flow simulations.

keywords: HPC, CFD, GPU, OpenFOAM, T-99.

Índice general

Resumen I

Abstract II

Índice general V

Índice de Figuras VII

Índice de Tablas VIII

Nomenclatura IX

1. La Dinámica de Fluidos Computacional del flujo en el tubo de aspiración. **1**

1.1. Simulación fluido dinámica 1

1.2. Simuladores de fluidos *CFD* 4

1.2.1. Caja de Herramientas OpenFOAM 6

1.2.2. Estructura de OpenFOAM 7

1.3. Energía hidroeléctrica y su relación con *CFD* 9

1.4. Tubo de aspiración de una turbina Kaplan 11

1.4.1. Antecedentes al estudio *CFD* en un tubo de aspiración 13

2. Cálculo de Alto Rendimiento usando *GPU* en *CFD* **14**

2.1. Cálculo en paralelo por medio de procesadores Gráficos 15

2.2. Herramientas para la implementación de GPGPU a OpenFOAM 17

2.3. Antecedentes al uso de GPU en CFD 20

3. Ante-Proyecto y problemática **21**

3.1. Planteamiento del problema 22

3.2. Justificación 23

3.3. Hipótesis . . . 24
3.4. Objetivos . . . 25
3.4.1. Objetivo general . . . 25
3.4.2. Objetivos específicos . . . 25
3.5. Metodología . . . 25

4. Validación de resultados del tubo de aspiración de la turbina T-99 28
4.1. Características del modelo . . . 28
4.2. Condiciones de Frontera . . . 29
4.2.1. Modelo matemático de turbulencia $\kappa - \varepsilon$ estándar . . . 32
4.3. Mallas del Problema T-99 . . . 34
4.4. Resultados en las secciones de Medición . . . 36

5. Pruebas de *HPC* y resultados cuantitativos computacionales 41
5.1. Equipo de cálculo . . . 42
5.2. Algoritmo de OpenFOAM a acelerar . . . 45
5.3. Resultados computacionales . . . 46
5.4. Benchmark computacional en Fluent . . . 54

Conclusiones de la validación CFD contra la experimental 58

Conclusiones del BenchMarck Computacional 59

Trabajo a Futuro 61

Conclusión Final 63

Bibliografía 64

Apéndice A 70
I. Geometría T-99 . . . 70
II. Malla OpenFOAM . . . 72

Apéndice B 76
I. Configuración para los solucionadores . . . 76
II. Configuración para los esquemas numéricos . . . 79
III. Configuración para el control de la base de datos . . . 80
IV. Configuración para las condiciones de frontera de épsilon . . . 82

V. Configuración para las condiciones de frontera de kappa 83
VI. Configuración para las condiciones de frontera de presión 84
VII. Configuración para las condiciones de frontera de velocidad 85
VIII. Codigo para ejecutar *Solver* en *CUDA* 85

Índice de figuras

1.1. a) Claude Louis Navier y b) George Gabriel Stokes. 2
1.2. Estructurado OpenFOAM 8
1.3. Tubo de aspiracion de una turbina hidráulica conectado al rodete 12

2.1. Diagrama de tareas computacionales en a) secuencial y b) paralelo [39]. . . 15
2.2. GPU + CPU, Combinacion Potente [39]. 17
2.3. Proceso CUDA de comunicación del *Host* (*CPU*) con la *GPU* [18]. 18

4.1. Esquema del tubo de aspiración de la turbina T-99 de [7] 29
4.2. Secciones de medición planteadas para el T-99 de [7] 31
4.3. Línea de medición a lo largo de la pared superior e inferior del tubo de aspiración[8]. 32
4.4. Mallado de las paredes en ICEM de ANSYS, cortesia la Universidad de LULEA 35
4.5. Grafica comparativa de Cpr en la sección superior e inferior respectivamente. 37
4.6. a) Líneas de corriente desde el cono de expansión y contornos de velocidad para las secciones Ib, II, III, IV y IVb arrojados por OpenFOAM (post-procesamiento en ANSYS©) en comparación al trabajo b) Líneas de corriente desde el cono de expansión y contornos de velocidad para las secciones por Ib, II, III, y IVb por M.J. Cervantes, [26]. 38
4.7. Contornos de velocidad cerca del cono de expansión del presente trabajo en comparación al trabajo realizado por M.J. Cervantes, [26], respectivamente. 39
4.8. Contornos de velocidad normal a la sección II (CS_II). 40
4.9. Contornos de velocidad horizontal a la sección III (CS_III). 40

5.1. Supercomputadora para el cálculo del benchmark de la T-99 44
5.2. Dispositivo *GPGPU*; *TESLA C1060* 44
5.3. Gráfica de tiempos de cálculo para la malla de 1 millón de elementos 49

5.4. Gráfica de tiempos de cálculo para la malla de 2 millones de elementos . . . 50
5.5. Gráfica de tiempos de cálculo para la malla de 6 millones de elementos . . . 51
5.6. Aceleración contra el número de nodos 52
5.7. Eficiencia unitaria del paralelismo contra el número de nodos (caso base 2 CPU) 53
5.8. Porcentaje de ahorro de tiempo computacional en base al cálculo más rápido 54
5.9. Gráfica de tiempos de cálculo para la malla de 1 millón de elementos en FLUENT 55
5.10. Gráfica de tiempos de cálculo para la malla de 2 millones de elementos en FLUENT 56
5.11. Aceleración contra el número de nodos en FLUENT 56
5.12. Eficiencia unitaria del paralelismo contra el número de nodos en FLUENT . 57
A1. Dimensiones del depósito de agua. El tubo de aspiración termina en un depósito cilíndrico grande y que tiene un diámetro de 2,600 mm, y en el centro de la salida del tubo de aspiración está situado a 282 mm por encima del centro del cilindro [7, 8]. 70
A2. Dibujos del tubo de aspiración T-99 [7, 8]. 71
A3. Localización de las secciones de medición [7, 8]. 72
A4. Salida texto de la aplicación checkMesh para la malla de 1 millón de elementos 73
A5. Salida texto de la aplicación checkMesh para la malla de 2 millones de elementos 74
A6. Salida texto de la aplicación checkMesh para la malla de 6 millones de elementos 75
B1. Configuración del caso para los *solvers (fvSolution)* 78
B2. Configuración del caso para los esquemas de discretización y numéricos (*fvSchemes*) 80
B3. Configuración del caso para el control de entrada y salida o también llamada I/O que establece parámetros esenciales para la creación de la base de datos de entrada como son las librerías. 82
B4. Configuración del caso para la disipación energética o épsilon (*epsilon*) . . . 83
B5. Configuración del caso para la energía cinética (k) 84
B6. Configuración del caso para el campo vectorial de presiones (P) 84
B7. Configuración del caso para el campo vectorial de velocidad (U) 85
B8. Código C para el método de gradiente bi-conjugado estabilizado con precondicionador diagonal (*cufflink_DiagPBiCGStab*) 89
B9. Código C para el método de gradiente conjugado (*cufflink_DiagPBiCGStab*) 92

Índice de tablas

3.1. Metodología planteada para llevar a cabo el proyecto 27

4.1. Velocidades que fueron establecidas como condiciones de frontera en la entrada o bien en la sección CS_Ia de [7]. 30

5.1. Tabla de propiedades principales del equipo de trabajo 42
5.2. Especificaciones de la TESLA C1060 43
5.3. Relación de casos para el benchmark 46
5.4. Tabla de tiempos computacionales para la malla de 1 millón de elementos . 48
5.5. Tabla de tiempos computacionales para la malla de 2 millones de elementos 49
5.6. Tabla de tiempos computacionales para la malla de 6 millones de elementos 51

Nomenclatura

AEA autoridad de la Energía Atómica.

ANSYS corporacion que desarrolla, comercializa y presta soporte a la ingeniería a través de software de simulación.

antialiasing se le llama antialiasing a los procesos que permiten minimizar el el efecto que causa que señales continuas digitales.

API interfaz de programación de aplicaciones (IPA) o API (del inglés Application Programming Interface).

benchmark es técnica usada para realizar una comparación informática de su potencial.

BIOS es el Sistema Básico de Entrada/Salida (Basic Input-Output System), conocido simplemente con el nombre de BIOS, es un programa informático inscrito en componentes electrónicos de memoria Flash existentes en la placa base.

bug es un error de software, comúnmente conocido como bug (bicho), es un error o fallo en un programa de computador o sistema de software que desencadena un resultado indeseado.

C y C++ lenguajes de programación diseñados para la manipulación de objetos.

CAD de su acrónimo anglosajón Computer-aided design o bien el diseño asistido por computadora ej. de software CAD autoCAD.

CFD o DFC de su acrónimo anglosajón Computational Fluid Dynamics o bien dinámica de fluidos computacional.

CFX software simulador de comportamiento fluido dinámico de ANSYS.

CFX-4 software de simulado de fluidos ahora de ANSYS.

Chip es un circuito integrado (CI), también conocido como chip o microchip, es una pastilla pequeña de material semiconductor, sobre la que se fabrican circuitos electrónicos.

CPU de su acrónimo anglosajón Central Process Unit; es el procesador central de un ordenador.

cpu-time es el tiempo de CPU (o uso del CPU, o tiempo de proceso) es la cantidad de tiempo en que la unidad central de proceso fue usada para procesar las instrucciones de un programa de computadora, en oposición a la espera por las operaciones de entrada/salida.

CUDA siglas de Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo).

default valores predeterminados de un programa si un programa lo requiere estos valores serán devueltos.

DirectX es una colección de API desarrolladas para facilitar las complejas tareas relacionadas con multimedia, especialmente programación de juegos y vídeo, en la plataforma Microsoft Windows.

DNS de sus siglas Direct numerical simulation (DNS) simulación fluido computacional que resuelve las ecuaciones de Navier-Stokes sin modelo de turbulencia.

ESI Group de su acrónimo anglosajón Experts in Virtual Product Engineering to support Industrial Innovation, corporación dedicada a brindar servicios y soportes tecnológicos.

execution time tiempo de ejecución total del programa.

FINE codigo de desarrollo por la empresa NUMECA.

FLOW-3D es un software que contiene amplias capacidades de modelado físico pionero en simulaciones 3D.

FLUENT es un software que contiene amplias capacidades de modelado físico necesario para modelar el flujo.

Fortran lenguaje de programación alto nivel de propósito general.

GPGPU procesador de Gráficos de propósito General.

GPL de su acrónimo anglosajón general public license o licencia publica general.

GPU de su acrónimo anglosajón General Process Unit; es el procesador de gráficos de un ordenador o computadora.

hardware partes tangibles de un sistema informático o sus componentes físicos estas son la parte medular pues si son de altas prestaciones realizaran las tareas rápidamente.

LANL laboratorio nacional de Los Alamos de sus siglas en ingles.

LES de sus siglas Large eddy simulation, técnica para simular flujos turbulentos usando la aplicación de Kolmogorov.

LP gas licuado del petróleo es la mezcla de gases licuados presentes en el gas natural.

N-S abreviacion de Navier-Stokes.

NASA Administración Nacional de la Aeronáutica y del Espacio y por sus siglas en inglés: National Aeronautics and Space Administration.

OpenFOAM de su acrónimo anglosajón Open Field Operation and Manipulation, software dedicado a la simulación de mecánica de medios continuos.

PHOENICS software pionero de capacidades de modelado físico.

RAM memoria principal de un ordenador.

scripts guion, archivo de órdenes o archivo de procesamiento por lotes y realiza una serie de tareas que el usuario requiera y programado.

sdk siglas de Samples Development Kit, kit de desarrollo de programación.

software equipamiento lógico o soporte lógico de un sistema informático y realiza las operaciones lógicas del usuario.

solver solucionador numérico computacional (soluciona los métodos numéricos).

STAR-CD software libre de capacidades de modelado físico.

TASCflow software de modelado físico necesario para modelar el flujo.

Upwind es un método de discretización para resolver ecuaciones diferenciales parciales hiperbólicas.

wall-clock time es el tiempo transcurrido del reloj interno o tiempo real este se traduce como tiempo de reloj de pared.

Workshop es el taller de investigación cientifica.

WSPAC estación de trabajo propiedad de la facultad de ingeniería mecánica UMSNH, sus siglas (WorkStation Provided and accelerated with Api Cuda).

Capítulo 1

La Dinámica de Fluidos Computacional del flujo en el tubo de aspiración.

Al observar a nuestro alrededor es posible contemplar múltiples fenómenos donde un fluido se involucra, desde el agua corriendo en el rio, el aire en el que planea un ave; inclusive en nuestros hogares en las tuberías del agua o gas LP . Es por ello que se sabe que el estudio en la ingeniería de estos fenómenos es de gran importancia. Esto ha llevado a científicos a formular las ecuaciones gobernantes de los fluidos. A la par, la necesidad energética ha ido en aumento y dispositivos que involucran una mejora y que son de comportamiento fluido dinámico requieren de la modelación usando la solución de dichas ecuaciones gobernantes. Estas ecuaciones han llevado a aproximaciones a lo real por medio de ordenadores, ordenadores que necesitan cada vez más potencia debido a la necesidad de resultados más confiables y con incertidumbre mínima.

1.1. Simulación fluido dinámica

Desde tiempos antiguos, ya Arquímedes analizaba el comportamiento de fluidos como el comportamiento del agua. Observó, entre otras cosas, cómo un fluido sometido a presión se desplaza desde la zona de mayor presión hasta la de menor. Fué ya Leonardo Da Vinci en el siglo XV quien realizó grandes contribuciones al estudio del comportamiento de los fluidos mediante el planteamiento de ecuaciones matemáticas. Uno de los trabajos más destacados fue la creación de la ecuación de continuidad o principio de conservación de masa.

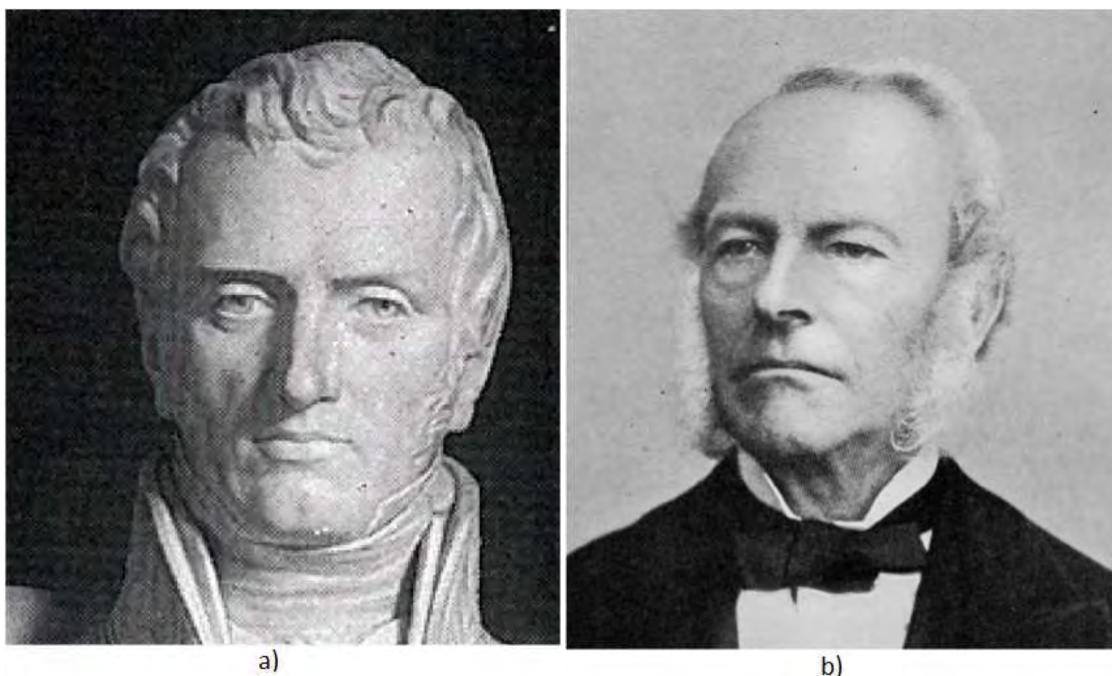

Figura 1.1: a) Claude Louis Navier y b) George Gabriel Stokes.

Pasando por valiosos contribuyentes al desarrollo de la Mecánica de Fluidos, a finales del siglo XIX, fueron Claude-Louis Navier y George Stokes (Figura 1.1) los que formularon teorías sobre la fricción interna de fluidos de movimiento y derivaron la famosa ecuación de Navier-Stokes. Se trata de un conjunto de ecuaciones en derivadas parciales no lineales que describen el movimiento de un fluido. Al solucionar esta ecuación se puede conocer el valor de las variables de un fluido tales como velocidad, presión, etc.[1].

Actualmente no se dispone de una solución general analítica para este conjunto de ecuaciones, salvo ciertos tipos de flujo y situaciones muy concretas. Debido a esto se debe recurrir al análisis numérico (utilizar computadores para realizar millones de cálculos) para determinar una solución aproximada. A la rama de la Mecánica de Fluidos ocupada de la obtención de estas soluciones mediante computadores se denomina Dinámica de Fluidos Computacional *(CFD ó Computational Fluid Dynamics)*. La Dinámica de Fluidos Computacional es una de las ramas de la Mecánica de Fluidos que usa métodos numéricos y algoritmos para estudiar y analizar problemas que involucran fluidos en movimiento, mediante la solución de las ecuaciones de Navier-Stokes, transferencia de calor e incluso otras (reacciones químicas) en

el computador. Existen diferentes métodos numéricos y algoritmos que resuelven de distinta forma las ecuaciones fundamentales. En otras palabras, la Dinámica de Fluidos Computacional es el arte de remplazar los sistemas de ecuaciones diferenciales parciales en un sistema algebraico de ecuaciones que pueden ser resueltas usando computadores. Esta proporciona información cualitativa y cuantitativa de la predicción del flujo de fluido por medio de la solución de las ecuaciones fundamentales usando métodos numéricos. La *CFD* permite a los científicos e ingenieros desarrollar "experimentos numéricos" (simulaciones computacionales) en un "laboratorio virtual" (ordenador). Estos experimentos permiten predecir comportamientos y conocer datos de las variables involucradas en el proceso. La información generada es importante y clave para comprobar el diseño y mejorar la eficiencia y comportamiento de un fenómeno estudiado.

Usando *CFD* es posible construir un modelo computacional que represente un sistema o equipo que se quiera estudiar. Después se especifican las condiciones físicas y químicas del fluido al prototipo virtual y el *software* entregará la predicción de la dinámica del fluido. La herramienta *CFD* ofrece la capacidad de simular flujos de gases, líquidos, transferencia de masa y calor, cuerpos en movimiento, física multi-fases, reacciones, la unidad interacción fluido-estructura y acústica a través de la modelación en el computador.

Una representación del fenómeno en *CFD* modelo-numérico está compuesta por 3 etapas: pre-procesamiento, procesamiento y post-procesamiento. Comienza con la modelación geométrica *CAD* del dominio a analizar, es decir representar en el computador la geometría de lo que se quiere simular. Este domino representa el fluido sobre el cual se quiere tener información, analizar y predecir el comportamiento. Actualmente existen diversos *softwares* comerciales especializados únicamente en generar modelaciones *CAD* . Una vez que se tenga el dominio computacional o modelo *CAD* del fluido, éste se discretiza o divide espacialmente en celdas para formar una malla. Las mallas pueden ser regulares, definidas por celdas en formas de triángulos (2D) o en tetraedros (3D), o pueden ser regulares definidas por celdas en formas de cuadrado (2D) o hexaedros (3D). Las propiedades físicas del fluido, tales como temperatura o velocidad, son configuradas según el problema, teóricamente hablando, estas son calculadas en cada uno de estos volúmenes como solución de las ecuaciones fundamentales, recordando que a un mayor número de elementos de la malla se necesitará mayor capacidad computacional. Luego de dividir el modelo geométrico en celdas se procede a generar la configuración de la simulación. Esto significa establecer materiales, velocidades en el contorno de la geometría, modelos adicionales para el análisis, etc. Una vez terminada la etapa de pre procesamiento continúa la etapa de solución de las ecuaciones. Los tiempos de cómputo dependen de varios factores: número de elementos, especificaciones del equipo usado, confi-

guración de la simulación. Posterior a la solución se procede a analizar los resultados, sacar conclusiones y por lo general a volver a ejecutar otra simulación.

La historia de la *CFD*, como es lógico, va ligada a la evolución de los ordenadores. Surgió en la década de los años 70's como medio para simular fluidos en movimiento, aunque para situaciones muy simples en aplicaciones aeroespaciales e industriales donde la predicción del comportamiento del flujo era importante. A mediados de la década de los 80's se fue desplazando el interés hacia los fluidos viscosos y por tanto hacia la resolución completa de la ecuación de Navier-Stokes. En los 90's el uso de la *CFD* se ha expandido de forma significativa a distintas aplicaciones y procesos industriales en los que interviene transferencia de calor, reacciones químicas (como combustión), flujos bifásicos, cambios de fase, transferencia de masa, y turbulencia. Con la evolución de los supercomputadores y con el desarrollo de nuevas técnicas numéricas, los problemas que se pueden resolver cada vez son más complejos[1].

1.2. Simuladores de fluidos *CFD*

En la década de 1930, las irresolubles limitaciones de los estudios analíticos fueron una motivación muy importante para continuar con el lento desarrollo de metodologías computacionales. Así, la primera simulación numérica (aún sin computadores) del flujo alrededor de un cilindro fue realizada en 1933 en Inglaterra por A. Thom y comunicada por el propio G.I. Taylor. Similares resultados fueron obtenidos por M. Kawaguti en Japón, resolviendo las ecuaciones de Navier-Stokes para el flujo alrededor de un cilindro para un número de Reynolds de 40. Según palabras del propio Kawaguti, “la integración numérica de este estudio requirió de un año y medio, con 20 horas de cálculo por semana y una considerable cantidad de trabajo y resistencia”. No en vano, él mismo realizó todos los cálculos, usando simplemente una calculadora de escritorio.

A partir de finales de la década de 1950 y en toda la década de los 60, el programa *LANL*, auspiciado por la *NASA* , se constituyó en el verdadero impulsor de las técnicas *CFD*, desarrollando los primeros códigos y dando los primeros pasos en el empleo de computadores.

A finales de los años 80's y principios de los 90's se produce un “boom” en la creación de códigos comerciales con los pioneros *PHOENICS* y *FLUENT*. Entre ellos, cabe destacar a *FIDAP* (*Fluid Dynamics Analysis Package*), desarrollado por *Illinois Institute of Technology* al mismo tiempo que *FLUENT* y que posteriormente FIDAP fue absorbido por *FLUENT*. Otro código interesante; *STAR-CD* desarrollado en 1987 como *spin-off* o bien subproducto de los trabajos del *Imperial College* y centrado en mallas móviles, no estructuradas; *FLOW3D*

desarrollado por la *AEA* en Inglaterra a finales de los años 80 para aplicaciones nucleares; con guión, a diferencia del anterior surgió de los esfuerzos del *LANL* en 1985 para comercializar su potente modelo de flujos con superficie libre (*SOLA-VOF*); *TASCflow*, desarrollado en Canadá también en 1985, y posteriormente fusionado con el *FLOW3D* de la *AEA* para dar lugar al *CFX-4* a mediados de los 90's o los códigos *FINE* (*Flow Integrated Environments*), desarrollados desde 1992 por *NUMECA*, empresa impulsada por Charles Hirsch de la Universidad Libre de Bruselas y autor de otro de los libros de referencia sobre *CFD* (*Hirsch, 1990*).

Las empresas aeronáuticas y automovilísticas no permanecen ajenas a estos cambios y comenzaron a utilizar códigos comerciales en sus fases de diseño. La aparición de mallas no estructuradas permite simulaciones con un grado de detalle que hasta entonces no era viable. Así, Boeing comienza a utilizar códigos comerciales desde 1996, primero con *CFD++* y después con *FLUENT* en 1998. No tardan en seguir sus pasos otras empresas como Airbus, utilizando Flowmaster, así como otras compañías del sector dedicadas a la fabricación de motores de avión, como General Electric, Pratt&Whittney o Rolls-Royce. A partir de 1995, las técnicas *CFD* comienzan a ser aplicadas en el sector del automóvil. Multitud de casos, tanto de flujos externos como de flujos internos, son analizados desde esta nueva óptica numérica. *General Motors* y *Ford* son pioneros en el uso de estas técnicas. En general, la aerodinámica del vehículo, en términos de fuerza de arrastre y penetración al aire, es el tema estrella de estas aplicaciones.

Debido a la complejidad de las ecuaciones y al grado de detalle que normalmente se desea obtener de las simulaciones, los ingenieros encargados en el desarrollo de estas técnicas *CFD* siempre han necesitado de las máquinas y supercomputadores más potentes del mundo. Por tanto, aunque se utilice un *software* comercial a nivel de usuario, siempre hay que recordar que estos códigos están basados en un conjunto de ecuaciones no lineales, muy complejas y acopladas entre sí, que se resuelven de forma iterativa mediante algoritmos muy específicos incluidos en el propio paquete (el *solver*).

El objetivo que se persigue es que el usuario sea capaz de resolver cualquier flujo dentro de una geometría prefijada, que se limita con unas condiciones iniciales y de frontera con el fin de que los fenómenos físicos implicados estan identificados a priori. Los resultados del código *CFD* comercial pueden normalmente representarse gráficamente o como un mapa de distribuciones, tanto de variables escalares (contornos) como de variables vectoriales (mapa de vectores, líneas de corriente, etcétera). La posterior evolución de todos estos códigos ha estado marcada por una intensa competitividad y una febril actividad de fusiones y cambios de manos. Tras el dominio inicial de *PHOENICS*, *FLUENT* se convirtió en el referente de

esta industria, sobre todo a partir de la segunda mitad de la década de los 90's. Sin embargo, desde 2003, se ha producido una importante concentración de estos paquetes, iniciada por *ANSYS* con la compra de *CFX-4*. ANSYS, líder mundial en el desarrollo de herramientas de análisis en el campo de la ingeniería asistida por computador (*CAE*), desembarcaba por fin en la industria del *CFD* con la adquisición de un importante código numérico, al que pasaría a llamar simplemente *CFX*.

Finalmente, tras unos años de intensa rivalidad comercial, en 2006 *ANSYS* compraba a *FLUENT* por un total de 630 millones de dólares, creando así al nuevo líder mundial en desarrollo de técnicas numéricas. Actualmente, sólo *CD-adapco*, heredera del código *STAR-CD* y participada por la importante empresa de *CAD Dassault Systems* (fabricante de *CATIA*), se puede comparar en cierto modo al nuevo gigante de ANSYS. La concentración de los códigos ha dado lugar a un progresivo incremento en los precios de las licencias. Sin embargo, esta tendencia puede verse contrarrestada con la aparición de nuevos códigos numéricos de distribución libre, siguiendo el espíritu de sistemas operativos libres, como Linux. El pionero en la creación de códigos libres de *CFD* es *OpenCFD*, que desarrolló un avanzado *solver* libre denominado *OpenFOAM*. Probablemente, en un futuro no muy lejano, el papel del *software* libre tenga un claro impacto en un campo tan especializado como el del *CFD*, obligando a reducir los precios de las licencias de los actuales distribuidores y volviendo en cierto modo a aquellos románticos tiempos a mediados de los sesenta en los que todo era un territorio nuevo por explorar [37].

En la actualidad existe una diversidad de simuladores de fluidos, algunos sólo están especializados para la visualización de resultados, otros para realizar únicamente el *CAD* de la geometría del problema a estudiar, otros más realizan los mallados y hay otros simuladores que contienen todas la herramientas necesarias para una simulación completa, este trabajo hace uso de recursos libres. *OpenFOAM* es una caja de herramientas completa para la simulación que ha estado surgiendo en los últimos años, herramientas para el pre-procesamiento, procesamiento y post-procesamiento.

1.2.1. Caja de Herramientas OpenFOAM

Este proyecto hace uso de un simulador de fluidos de nombre *OpenFOAM* de *ESI Group* que ofrece al usuario una caja de herramientas para realizar estudios fluido dinámicos, surgió desde el año 2004 bajo Licencia Publica General (*GPL*). Su lenguaje de compilación y programación es C++, siendo requerido conocimientos base de este lenguaje para poder realizar la configuracion correcta de un problema. Tiene una gran base de usuarios en la

mayoría de las áreas de la ingeniería y de la ciencia, tanto de las organizaciones comerciales y académicas. *OpenFOAM* cuenta con una amplia gama de características para resolver cualquier cosa, desde los flujos de fluidos complejos que involucran reacciones químicas, la turbulencia y la transferencia de calor, a la dinámica de sólidos y electromagnetismo. Incluye herramientas para mallado, en particular *snappyHexMesh*, un mallador paralelizado para geometrías *CAD* complejas, y para pre y post-procesamiento. Se ejecuta en paralelo o en serie, lo que permite a los usuarios sacar el máximo partido del *hardware* de ordenador a su disposición. Al ser abierto, *OpenFOAM* ofrece al usuario total libertad para personalizar y ampliar la funcionalidad existente. Sigue un diseño de código muy modular con múltiple funcionalidad (por ejemplo, métodos numéricos, mallado, modelos físicos, etc.) se compilan cada uno en su propia librería compartida. Aplicaciones ejecutables que se crean con ligarlo simplemente a la funcionalidad de una librería. *OpenFOAM* incluye más de 80 aplicaciones que simulan los *solver* a problemas específicos de la ingeniería mecánica y más de 170 aplicaciones de utilidades que realizan tareas previas y posteriores al procesamiento, por ejemplo, mallado, visualización de datos, etc. El presente trabajo hace uso de este código debido a su paralelización, disponibilidad de código y que resuelve ecuaciones donde implica turbulencia y que es modelada a un dispositivo de desfogue usado en plantas de generación energética por medio de la Hidrodinámica.

1.2.2. Estructura de OpenFOAM

Los datos de un caso se guardan en una serie de directorios estructurados en contraste a la mayoría de los programas *CFD*. En la figura 1.2 se encuentra la configuración que debe de seguir un caso a estudiar. La guía de usuario [38] nos proporciona información acerca de cada uno de los *scripts* para un caso en particular, además cuenta con tutoriales en el repositorio para una mayor comprensión. Se enlista a continuación una breve explicación de los directorios que están involucrados en la resolución de un caso.

i) Directorio de Sistema: este directorio es usado para asociar los parámetros usados para la solución. Contiene al menos 3 archivos *controlDict* donde los parámetros de control se corren incluyendo los tiempos de empiezo y final, paso de tiempos y parámetros para datos de salida; *fvSchemes* donde esquemas de discretización serán usados en la solución al momento de ejecutarse y fvSolution donde las ecuaciones solucionadoras, tolerancias y otros algoritmos seleccionados para la simulación.

ii) Directorio de constantes: contiene una descripción completa de la malla en un subdirectorio llama *polyMesh* además de contener en archivos las propiedades físicas que se aplican

al problema, por ejemplo para propiedades de transporte el *script* es: *transportProperties*.

iii) Directorios de tiempo: Este directorio contiene archivos individuales y datos para casos particulares. Estos datos pueden ser condiciones iniciales y condiciones de borde que el usuario debe especificar al definir el problema o resultados que han sido escritos por *OpenFOAM*. Es de notar que el programa debe de inicializarse en el tiempo aunque la solución no lo requiera, como es el caso del estado estable. El nombre de cada directorio se basa en el tiempo de la ejecucion, por ejemplo para las condiciones inciales el directorio se llama 0 y debe establecerse como condición de entrada de datos.

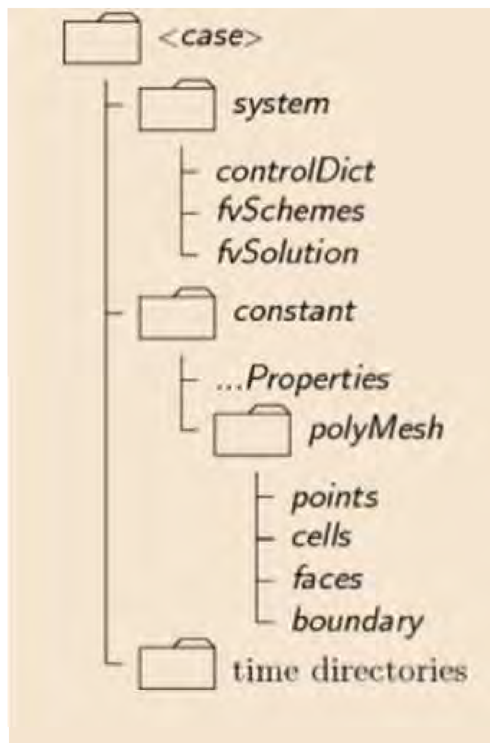

Figura 1.2: Estructurado OpenFOAM

Cabe notar que *OpenFOAM* resuelve directamente los casos en 3D por *default* pero se puede instruir al programa para resolver problemas en 2D por medio de una especificación de condición de frontera "vacia". El *software OpenFOAM®* es muy usado académicamente al contar con código abierto, pues se puede llevar a cabo un trabajo para la validez de datos experimentales hasta explorar científicamente y desarrollar el código más ampliamente. Contar con un programa de cómputo *(software)* académico facilita el aprendizaje del software *CFD* comercial, pues es más complejo que el *Software* comercial de utilizar, siendo también esta una desventaja. *OpenFOAM* cuenta con la herramienta de pre-procesamiento *blockMesh* que

genera mallas desde una descripción especificada en un diccionario de entrada, *blockMeshDict* que a su vez está localizado en el directorio de *constant/polymesh*. Además cuenta con otra herramienta para el post-procesamiento el *ParaFoam* que también puede ser utilizado para checar el modelado. En cuanto al modelado puede ser usado por medio de generar la geometría en texto, en particular *OpenFoam* permite importar y exportar geometría. El uso del calculo por medio de computadores en paralelo es también una opción de *OpenFOAM*, su uso es por medio de dividir su dominio físico a subdominios.

1.3. Energía hidroeléctrica y su relación con *CFD*

El deseo político de reducir el uso de energía nuclear y los combustibles fósiles se ha ido incrementando en muchas partes de Europa y del resto del mundo, donde probablemente hará falta la electricidad. Es de preocupación mundial la contaminación debido a los gases del efecto invernadero y sobre todo a la necesidad de un futuro sustentable de energía. Esta sustentabilidad puede ser compensada con fuentes de energía que se consideran renovables, como la eólica, la solar y la energía hidráulica. Al aumentar el uso de energía solar y la energía eólica, la energía hidráulica es necesaria como reserva para los días nublados y de tranquilidad pero también como un compensador de fase, puede concluirse que la energía del agua se ha vuelto en un proveedor de electricidad indispensable.

La energía hidráulica es la fuente renovable de electricidad más importante y más utilizada en el mundo. Representa un 19 % de la producción total de electricidad, siendo Canadá el productor más importante de energía hidroeléctrica, seguido por los Estados Unidos y Brasil. Aproximadamente dos tercios del potencial hidroeléctrico económicamente viable quedan aún por desarrollar (*World Energy Council* 2010). La energía hidráulica no aprovechada es todavía muy abundante en América Latina, África central, India y China. La energía hidráulica tiene la cualidad de ser renovable, pues no agota la fuente primaria al explotarla, y es limpia ya que no produce en su explotación sustancias contaminantes de ningún tipo. Sin embargo, el impacto medioambiental de las grandes presas, por la severa alteración del paisaje es considerable e incluso se ha llevado la inducción de un microclima diferenciado en su emplazamiento y que ha desmerecido la bondad ecológica de este concepto en los últimos años. Al mismo tiempo, la madurez de la explotación hace que en los países desarrollados no queden apenas ubicaciones atractivas por desarrollar nuevas centrales hidroeléctricas, por lo que esta fuente de energía, que aporta una cantidad significativa de la energía eléctrica en muchos países no permite un desarrollo adicional excesivo. Recientemente se están realizando

centrales minihidroeléctricas, mucho más respetuosas con el ambiente y que se benefician de los progresos tecnológicos, logrando un rendimiento y una viabilidad económica razonables. La energía hidráulica se basa en aprovechar la caída del agua desde cierta altura. La energía potencial, durante la caída, se convierte en cinética. El agua pasa por las turbinas a gran velocidad, provocando un movimiento de rotación que finalmente se transforma en energía eléctrica por medio de los generadores.

Este recurso energético se encuentra de forma natural disponible en las zonas que presentan suficiente cantidad de agua y una vez utilizada, es devuelta río abajo. Su desarrollo requiere construir pantanos, presas, canales de derivación y la instalación de grandes turbinas y equipamiento para generar electricidad. Todo ello implica la inversión de grandes sumas de dinero, por lo que no resulta competitiva en regiones donde el carbón o el petróleo son baratos. Sin embargo, el peso de las consideraciones medioambientales y el bajo mantenimiento que precisan una vez que estén en funcionamiento, centran la atención en esta fuente de energía.

Para el proceso de transformación de energía es necesario Turbomáquinaria. Una turbomáquina consta fundamentalmente de una rueda de álabes, rodete, que gira libremente alrededor de un eje cuando pasa un fluido por su interior. La forma de los álabes es tal que cada dos álabes consecutivos forman un conducto que obliga al flujo a variar su cantidad de movimiento, lo que provoca una fuerza, esta fuerza que al desplazarse el álabe provoca un trabajo. La clasificación fundamental de una turbina (convierte la energía del flujo en una energía mecánica en el eje, lo contrario seria una bomba hidráulica) es las de acción y las de reacción [2] .

- Turbinas de acción: Se llaman así cuando la transformación de la energía potencial en energía cinética se produce en los órganos fijos anteriores al rodete (inyectores o toberas). En consecuencia el rodete sólo recibe energía cinética. La presión a la entrada y salida de las cucharas (o álabes) es la misma e igual a la atmosférica.

- Turbinas de reacción: Se llama así cuando se transforma la energía potencial en cinética íntegramente en el rodete. La presión de entrada es muy superior a la presión del fluido a la salida. Otra clasificación muy distinta es en función de la dirección del flujo en el rodete, lo que puede hacer que clasifiquemos a las turbomáquinas en:

- Axiales: El desplazamiento del flujo en el rodete es paralelo al eje. Es axial y tangencial (giro).

- Radiales: El desplazamiento en el rodete es perpendicular al eje. No tiene componente axial.
- Mixtas: Tiene componente Axial, radial y tangencial.

En la actualidad, las turbinas que dominan el campo en las centrales hidroeléctricas son:

- Pelton (de acción)
- Francis (de reacción)
- Hélice y Kaplan (de reacción)
- Bulbo (de reacción)

Muchas plantas de energía, especialmente en los países de bajos recursos, están envejeciendo y necesitan ser actualizadas. La mayoría de ellas utilizan turbinas de tipo Kaplan. Una actualización a menudo implica un diseño nuevo de los rodetes y ya que existe una fuerte interacción entre el flujo en el canal y el tubo de aspiración en turbinas Kaplan, hay una necesidad de entender cómo el agua fluye en el tubo de aspiración para diseñar un rodete eficiente.

1.4. Tubo de aspiración de una turbina Kaplan

El flujo alrededor del eje de una turbina tipo Kaplan y en el cono de expansión es importante simular con precisión, ya que el 70 % de la recuperación de la presión se puede producir en esta región, de acuerdo a Jonsson, 2011 [5].

El tubo de aspiración recibe otros nombres, tales como hidrocono, difusor, etc. y consiste en una conducción, normalmente acodada, que une la turbina propiamente dicha con el canal de desagüe. Tiene como misión recuperar al máximo la energía cinética del agua a la salida del rodete o, dicho de otra forma, aprovechar el salto existente entre la superficie libre del agua y la salida del mismo, Fig. 1.3.

En su inicio, partiendo de la unión circular con la turbina, se trata de un conducto metálico que, en la mayoría de los casos, va aumentando gradualmente de diámetro, tomando forma tronco-cónica, tramo conocido como tubo de aspiración. Sobre el mismo se dispone, lateralmente, de una o dos entradas Figura 1.3, opuestas en el segundo caso, a fin de poder realizar revisiones, trabajos, etc. [3].

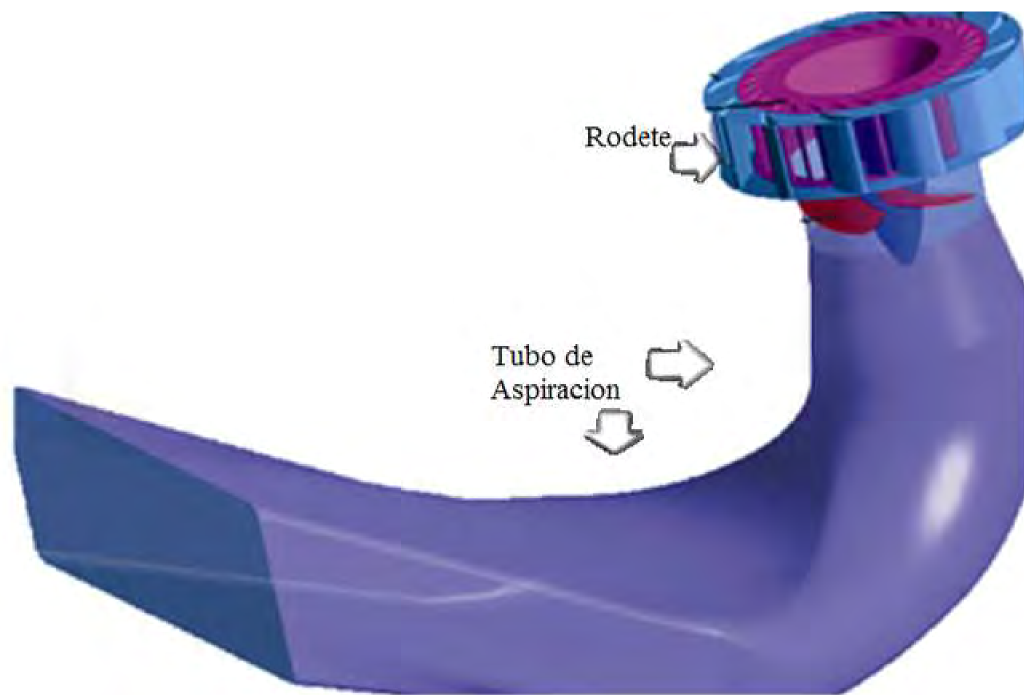

Figura 1.3: Tubo de aspiracion de una turbina hidráulica conectado al rodete

Al tener esa misión importante de recuperar energía, para centrales hidráulicas donde existan turbinas en serie; el dispositivo ha sido ampliamente estudiado por la comunidad de científicos. La eficiencia del tubo de aspiración es importante para la eficiencia de toda una planta de generación de energía y especialmente para una turbina Kaplan ya que la pérdida relativa es inversamente proporcional a la carga. 50 % de todas las pérdidas en una planta de energía, puede ocurrir en el tubo de aspiración de acuerdo con Andersson [4].

Una conclusión similar hace el mismo Andersson[4], donde afirma que el 80 % de la recuperación de la presión total sucede en el tubo de aspiración Kaplan y sobre todo en el primer 10 % del tubo de aspiración. Existe turbulencia en todo el tubo de aspiración, es por ello que para simular este importante dispositivo con resultados confiables, se necesitan modelos de turbulencia computacionalmente intensos así como mallas de gran densidad.

Ya que las mayores pérdidas en este dispositivo se generan por el desarrollo del flujo del fluido, la DFC o *CFD* es una herramienta que ha sido utilizado en los últimos años para estudiar a detalle este flujo. Sin embargo las limitaciones en la potencia de cálculo han hecho casi imposible la utilización de esquemas numéricos y modelos de turbulencia que logren

capturar los detalles del flujo. Así, para simular el flujo en este importante dispositivo con resultados confiables se requiere de la adaptación del Cálculo de Alto Rendimiento para lograr su paralelismo y así acelerar la solución.

1.4.1. Antecedentes al estudio *CFD* en un tubo de aspiración

Durante la última década se han realizado investigaciones sobre el tubo de aspiración de una turbina Kaplan ya sea de forma independiente o durante los talleres de la Turbina 99 realizados en Suecia. Algunos de estos trabajos se enlistan a continuación:

Los trabajos de Engström y colaboradores [6, 7, 8], son un compendio de los talleres realizados a través del modelo de un tubo de aspiración prototipo, los resultados de la simulación numérica fueron reportados por varios grupos de investigación que asistieron a un taller internacional, donde se discutió detalladamente para poder encontrar un modelo de turbulencia numéricamente adecuado y las variables que involucran al dispositivo que incluyen trabajos que van desde optimización de malla, tratamiento de la pared y hasta el uso de diferentes modelos de turbulencia de una Turbina-99 tipo Kaplan.

En los trabajos de Andersson [4, 13], se encuentran los resultados experimentales para este dispositivo, resultados que se usaron para validar experimentalmente el presente trabajo.

O. Petit [10], hace un estudio interesante sobre la diferencia entre *CFX* y *OpenFOAM*. Autores importantes como H. Nilsson y M. Page [9], tienen diversos estudios sobre este dispositivo, desde el análisis del perfil de entrada hasta simulaciones hechas sobre la plataforma *OpenFOAM*.

Galván S.R. y M. J. Cervantes [12, 26] también han contribuido importantemente en la parte de modelación como es el análisis de turbulencia y tratamiento de pared. Estos últimos trabajos mencionados han sido analizados especialmente para la validación numerica y la realización de este proyecto.

H. Nilsson en 2007[14] realizó una de las contribuciones más importantes en donde hace uso de ordenadores con cálculos en paralelo del tubo de aspiración de la Turbina T-99. Una conclusión importante es que la paralelización realizando la tarea de los cálculos en 16 núcleos del *CPU* (de su término anglosajón *Central Process Unit*) aumentó la eficiencia de 7.2 a 8.2 veces de la velocidad de cálculo, en contraste a usar un solo núcleo del *CPU*. La necesidad de cálculo de alto rendimiento es evidente y el presente trabajo está basado en esa problemática, haciendo uso de una nueva tendencia en cálculo de alto rendimiento; los procesadores gráficos como paralelizador de tareas.

Capítulo 2

Cálculo de Alto Rendimiento usando *GPU* en *CFD*

En la actualidad hay tres prácticas comunes de *CFD*, la forma más aproximada y completa es resolver las ecuaciones de Navier-Stokes por medio de discretización de sus ecuaciones. La segunda es una práctica que está basada en clases de partículas, ésta resuelve ecuaciones diferentes a las de Navier-Stokes por medio de sus interacciones microscópicas, en Hidrodinámica se usa generalmente el método *SPH* de su término anglosajón *Smoothed Particle Hydrodynamics* es decir partículas hidrodinámicas suavizadas. La tercera práctica es conocida como automatización celular, ésta es la solución más rápida computacionalmente hablando, sin embargo es también la menos exacta por su carencia en Matemáticas, Física y en sí de las bases científicas. En cuanto a tiempo computacional, la primera práctica requiere más tiempo computacional debido a su base científica y de la cantidad enorme de cálculos matriciales que requieren resolver los algoritmos de resolución, es por ello que se requiere una paralelización de tareas, la cual se puede realizar por medio de un procesador gráfico debido a sus cualidades como *hardware*. La unidad de procesamiento gráfico o *GPU* (acrónimo del inglés *graphics processing unit*) es un coprocesador dedicado al procesamiento de gráficos u operaciones de punto flotante, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos y/o aplicaciones *3D* interactivas. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la *GPU*, la unidad central de procesamiento (*CPU*) puede dedicarse a otro tipo de cálculos. En los últimos años ha resurgido la aplicabilidad del *GPU* en relación al paralelismo de tareas, el cálculo matricial en *CFD* es una de ellas, en especial a la resolución de las ecuaciones resultantes de la discretización de las ecuaciones de Navier-Stokes.

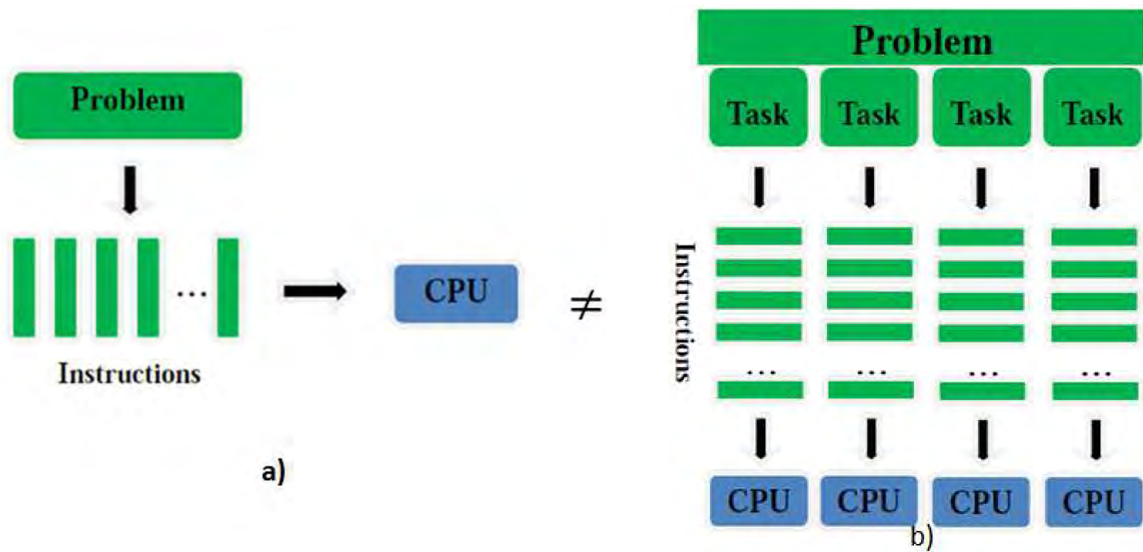

Figura 2.1: Diagrama de tareas computacionales en a) secuencial y b) paralelo [39].

2.1. Cálculo en paralelo por medio de procesadores Gráficos

La computación paralela es una técnica de programación en la que muchas instrucciones se ejecutan simultáneamente ver Figura2.1. Se basa en el principio de que los problemas grandes se pueden dividir en partes más pequeñas que pueden resolverse de forma paralela (no es lo mismo que concurrente). Existen varios tipos de computación paralela: paralelismo a nivel de bit, paralelismo a nivel de instrucción, paralelismo de datos y paralelismo de tareas.

Durante muchos años, la computación paralela se ha aplicado en la computación de altas prestaciones, pero el interés en ella ha aumentado en los últimos años debido a las restricciones físicas que impiden el escalado en frecuencia. La computación paralela se ha convertido en el paradigma dominante en la arquitectura de computadores, principalmente en los procesadores multinúcleo. Sin embargo, recientemente, el consumo de energía de los ordenadores paralelos se ha convertido en una preocupación.

La unidad de procesamiento gráfico o *GPU* es un coprocesador dedicado al procesamiento de gráficos. La *GPU* implementa ciertas operaciones gráficas llamadas primitivas optimizadas para el procesamiento gráfico. Una de las operaciones primitivas más comunes para el procesamiento gráfico en 3D es el *antialiasing* que suaviza los bordes de las figuras para darles un aspecto más realista. Adicionalmente existen operaciones primitivas para dibujar rectángulos, triángulos, círculos y arcos. Las *GPU* actualmente disponen de gran cantidad

de operaciones primitivas, buscando mayor realismo en los efectos.

Al inicio, la programación de la *GPU* se realizaba con llamadas a servicios de interrupción de la *BIOS*. Tras esto, la programación de la *GPU* se empezó a hacer en el lenguaje ensamblador específico a cada modelo. Posteriormente, se introdujo un nivel más entre el *hardware* y el *software*, con la creación de Interfaces de programación de aplicaciones (*API*) específicas para gráficos, que proporcionaron un lenguaje más homogéneo para los modelos existentes en el mercado. La primera *API* usada ampliamente fue el estándar abierto *OpenGL* (*Open Graphics Language*), tras el cual Microsoft desarrolló *DirectX*. Tras el desarrollo de esta *API*, se decidió crear un lenguaje más natural y cercano al programador.

Los *chips* gráficos empezaron como procesadores gráficos de funciones fijas, pero se hicieron cada vez más programables y potentes desde el punto de vista computacional, lo que permitió a NVIDIA introducir la primera *GPU* con propósitos distintos a la de renderizar pixeles. Entre los años 1999 y 2000, científicos del sector informático y de otras disciplinas empezaron a utilizar las *GPU* para acelerar diversas aplicaciones científicas. Fue el nacimiento de un nuevo concepto denominado *GPGPU* o *GPU* de propósito general. Aunque los usuarios conseguían un rendimiento sin igual (por encima de 100x con respecto a las *GPU* en algunos casos), el problema era que las *GPGPU* precisaban el uso de *API*s de programación de gráficos como *OpenGL* y *Cg (C for graphics)* al programar para las *GPU*. Eso limitaba el acceso a la enorme capacidad de las *GPU* en el campo científico. NVIDIA se dio cuenta del potencial que suponía aportar este rendimiento a toda la comunidad científica, de modo que invirtió para qué la *GPU* fuera totalmente programable y ofreció un proceso totalmente transparente para desarrolladores con lenguajes familiares como *C*, *C ++* y *Fortran* .

CUDA es una arquitectura de cálculo paralelo de *NVIDIA* que aprovecha la gran potencia de la *GPU* para proporcionar un incremento extraordinario del rendimiento del sistema. Los sistemas informáticos están pasando de realizar el "procesamiento central" en la *CPU* a realizar "coprocesamiento" repartido entre la *CPU* y la *GPU*. Para posibilitar este nuevo paradigma computacional, *NVIDIA* ha inventado la arquitectura de cálculo paralelo CUDA, que ahora se incluye en las *GPU*s *GeForce, ION, Quadro y Tesla*, lo cual representa una base instalada considerable para los desarrolladores de aplicaciones. *CUDA* ha sido recibida con entusiasmo por la comunidad científica. Por ejemplo, se está utilizando para acelerar *AMBER (The Assisted Model Building with Energy Refinement)*, un simulador de dinámica Molecular empleado por más de 60,000 investigadores del ámbito académico y farmacéutico de todo el mundo para acelerar el descubrimiento de nuevos medicamentos.

Una *CPU + GPU* constituye una potente combinación (Fig. 2.2) porque la *CPU* está formada por varios núcleos optimizados para el procesamiento en serie, mientas que la *GPU*

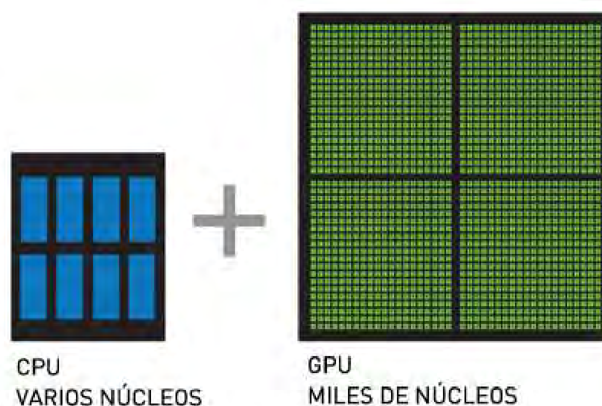

Figura 2.2: GPU + CPU, Combinacion Potente [39].

consta de millares de núcleos más pequeños y eficientes diseñados para el rendimiento en paralelo. Es evidente que si se utiliza para resolver los cálculos matriciales que implica la resolución de modelos *CFD* se optimizará ésta. Las partes en serie del código se ejecutan en la *CPU* mientras que las paralelas se ejecutan en la *GPU*. La comunidad científica lo ha aplicado en diferentes ramas de investigación y *CFD* es una de ellas.

2.2. Herramientas para la implementación de GPGPU a OpenFOAM

Dado que el dispositivo *GPU* no puede acceder directamente a la memoria principal de la computadora, los datos deben ser copiados o transferidos primeramente a la memoria incluida de la *GPU* con el fin de acceder a él. Esto también podría ser una ventaja en lugar de una limitación ya que su acceso a los núcleos de la *GPU* es rápido. La *GPU* no inicia el proceso hasta que la *CPU* ordena que lo haga, por lo tanto las instrucciones han de pasarse de la *CPU* primeramente. Una vez recibidas las instrucciones, el *GPU* utiliza sus múltiples núcleos para ejecutar el proceso en paralelo. El resultado de estos procesos en paralelo se almacena en la memoria en la *GPU*. Para que la *CPU* pueda interpretar los datos, se tiene que dar las instrucciones para copiar los resultados a la memoria principal de la computadora. La Fig.2.3 presenta las instrucciones típicas de un proceso *CUDA*.

Para llevar a cabo el procesamiento que hace *CUDA* pero aplicado a *CFD*, se harán por medio de librerías de interfaz y las más importantes son *CUBLAS*, *CUPSPARSE*, *thrust*, y

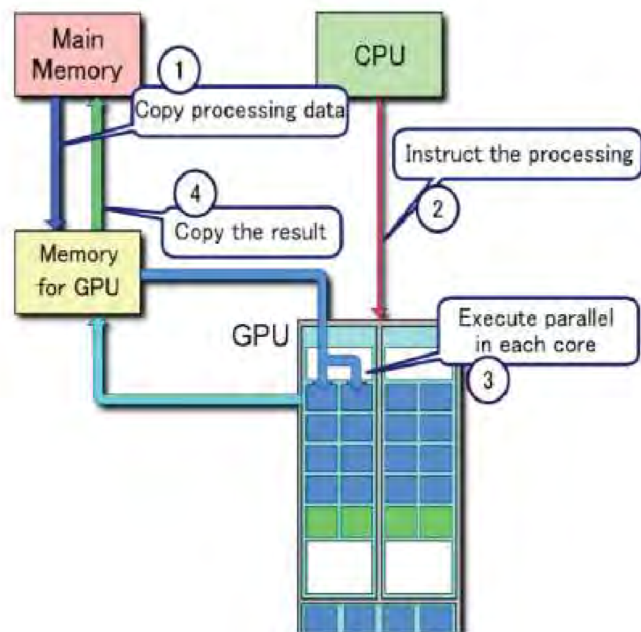

Figura 2.3: Proceso CUDA de comunicación del *Host* (*CPU*) con la *GPU* [18].

Cufflink library, se detallan a continuación:

CUDA es la tecnología que incluye una serie de aplicaciones de desarrollo Nvidia para comunicar con lenguaje de programación a la *GPU* con la *CPU*. La Fig. 2.3 muestra el código para realizar una tarea en paralelo por medio de la tecnología CUDA. *CUDA Sdk* contiene librerías que apuntan principalmente al objetivo del proyecto, éstas son *CUBLAS* y *thrust* (comunicación entre el *CPU* y el *GPU*).

La librería *CUBLAS* contiene subprogramas básicos de Álgebra Lineal. Esta librería es independiente a nivel de *API*. Cuenta con programas que realizan el producto escalar de dos vectores como entradas. En cuanto a multiplicaciones de matrices y vectores, contiene subprogramas que realizan operaciones de cualquiera de las combinaciones de la matriz-vector o combinaciones de matriz-matriz como entradas. La librería proporciona por lo menos 152 diferentes rutinas que manipulan los siguientes tipos de datos: float, double, complex y double complex. Con el fin de interconectar la librería *CUBLAS*, el código debe incluir el cublas.h. Desafortunadamente las funciones básicas dentro de esta librería no devuelven errores, pero la biblioteca también proporciona funciones para recuperar el último error registrado para un proceso de manejo de errores. Hay varias funcionalidades relativas a la asignación de memoria y transferencia de datos y que también se incluyen en esta librería. Es importante señalar también que esta librería utiliza como almacenamiento a la columna principal. Desde

la versión de *CUDA* 4.0, esta librería cuenta ahora con rutinas para agregar el manejo a los argumentos, dando como resultados el control necesario para gestionar múltiples hilos de ejecución y la capacidad de usar múltiples *GPU*s. Todas las funciones son ahora capaces de devolver un código de error, así como pasar y devolver valores escalares en función de memoria de la *GPU*.

Thrust es una librería única que mas que relacionarse con operaciones matriciales son plantillas para poder incluir archivos y cabeceras de *CUDA* en *lenguaje* $C++$ *STL*. Esto elimina la necesidad de hacer links. Con *thrust* se tiene iteradores para la albergación de datos y se definen los rangos donde el algoritmo se va a ejecutar. Esta librería contiene algoritmos de clasificación, reducción y análisis de rutinas. Estos algoritmos operan en un rango y soportan operadores de tipos general.

El *CUPSPARSE* o su abreviatura *Cusp* es la piedra angular para solucionadores lineales de matrices dispersas. Es la nueva librería para Álgebra Lineal dispersa básica. Esta librería contiene rutinas para convertir matrices densas a diferentes formatos como :

1. Lista de coordenadas (*COO)*
2. Fila dispersa comprimida (*RSE*)
3. Columna dispersa comprimida (*CSC*).

También *Cusp* incluye rutinas optimizadas para la multiplicación de matrices dispersas y vectores como entradas. Como ventaja es que con esta librería se puede implementar el Bi gradiente Conjugado Estabilizado (BiCGStab) para sistemas matriciales no simétricos. Tal solucionador requiere de memoria relativamente pequeña y con frecuencia tienen una convergencia más estable que otros solucionadores.

Cufflink (*CUDA for link Foam*) es una librería de código abierto para vincular los métodos numéricos basados en la Arquitectura unificada con lenguaje de programación C / C ++ de *NVIDIA* (*CUDA* ™) y *OpenFOAM* ®. En la actualidad, la librería utiliza los solucionadores lineales de *CUPS* y métodos de thrust para resolver el sistema de algebra dispersa lineal Ax = b de la clase *lduMatrix* de *OpenFOAM* y que devuelve el vector solución. *Cufflink* está diseñado para utilizar el paralelismo fino en OpenFOAM ® (a través de la descomposición de dominios).

2.3. Antecedentes al uso de GPU en CFD

La comunidad científica lo ha aplicado en diferentes aplicaciones y CFD es una de ellas trabajos publicados más cercanos a la simulación se presentan a continuación.

Los trabajos de Corrigan A.(et all), Stantchev G (et all). y Cohen J. (et all)[15, 16, 17], refieren a fenómenos en particular y que han requerido de programación casera de las ecuaciones diferenciales sin usar simuladores comerciales ni de código abierto.

A contraste del trabajo de Hai Tuan [18] donde realiza su trabajo sobre la plataforma de OpenFOAM donde su caso base es un fluido compresible. Una importante contribución se hace en [36] con aplicación a un fluido sanguíneo usando *solvers* básicos en el *GPU* apuntando al cálculo matricial.

Capítulo 3

Ante-Proyecto y problemática

Recientes trabajos de investigación en los últimos años, han comprobado cómo la potencia de cálculo del *GPU* supera a la del *CPU*. Sin embargo, ni los usuarios ni los programadores se han apresurado en remplazo ya que esta enorme potencia computacional tiene las siguientes restricciones [36]:

1. Los *GPU's* están diseñados como co-procesadores del *CPU* y no pueden ejecutar programas por sí mismos, ni comunicarse con cualquier dispositivo externo que no sea el *CPU* o bien el *host*.

2. El arreglo de los núcleos de *GPU* es una estructura jerárquica multi-niveles y los núcleos necesitan ejecutarse en el mismo programa, en nivel jerárquico desde el más bajo y deben ejecutar la misma instrucción o bien permanecer inactivos.

3. La memoria del *GPU* tiene una organización muy especial que funciona de manera óptima cuando se accede a los datos en orden, como si fuese una corriente de objetos consecutivos.

4. El *bus PCI-Express* que conecta *GPU's* a la *CPU* es relativamente lento (hasta 8 GB/s) es decir, mucho más lento que los buses internos del *CPU* y del *GPU*.

5. Los GPU's no pueden ejecutar programas escritos en lenguaje de programación para *CPU's* y que significa que prácticamente todo el software numérico para *GPU's* debe ser escrito desde cero.

Teniendo en cuenta estas limitaciones, se concluye que la aceleración eficaz que proporciona el *GPU* sólo es posible aplicarla a problemas con grandes conjuntos de datos homogéneos, que

pueden ser fácilmente procesados con al menos decenas de miles de *"threads"* independientes e idénticos los cuales compartan sólo de vez en cuando los datos vinculados entre ellos.

Para que un programa de aceleración a través del *GPU* sea eficiente debe de minimizar la transferencia de datos entre el *CPU* y el *GPU* y entre los núcleos del *GPU* y la memoria principal del *GPU*, para que el costo de esta transferencia se compensada por una gran cantidad de cálculos realizados en el *GPU*.

También está claro que la aceleración de simulaciones en *CFD,* las cuáles se basan en códigos complejos del *CPU* escritos en las últimas décadas por grupos de profesionales altamente calificados, deben llevarse a cabo en pequeños pasos dirigidos a pequeñas y específicas zonas que son críticas para el rendimiento total de los programas de *CFD*.

3.1. Planteamiento del problema

Teniendo en cuenta la problemática descrita anteriormente, en *CFD* los puntos 2 y 3 ya descritos son ventajas puesto que el orden de las matrices es jerárquico y se mandan miles e inclusive millones de órdenes de cálculo matricial en forma de objetos consecutivos. Los puntos 1,4 y 5 son los puntos medulares a resolver en el presente proyecto, esto se logrará con la teoría presentada en el capítulo anterior; resolviendo la comunicación entre el *host* (*CPU*) y el dispositivo (*GPU)* por medio de códigos escritos para procesadores gráficos para realizar cálculos. Los códigos *CFD* más populares que resuelven las ecuaciones *N-S* constan de tres pasos principales:

El espacio y tiempo continuo son aproximados por conjuntos finitos de puntos discretos (generación de la malla).

- Las ecuaciones diferenciales parciales que rigen el flujo se discretizan en los puntos de malla, lo que conduce a un gran conjunto de ecuaciones algebraicas lineales o no lineales.

- Por último, las ecuaciones algebraicas resultantes se resuelven utilizando métodos directos o iterativos.

En este último paso, se hace uso de métodos de linealización de las ecuaciones resultantes. La razón principal es el hecho de que las ecuaciones algebraicas lineales pueden ser resueltas mediante el uso de métodos computacionales muy eficientes de Álgebra Lineal.

Los sistemas lineales que se encuentran en problemas *CFD* tienen algunas características comunes:

- El número de incógnitas en estas ecuaciones es proporcional al número de vértices de malla, lo que resulta ser muy grande, a menudo inferior o igual a $1x10^6$.
- Cada ecuación incluye sólo las variables de vértices vecinos, y por lo tanto el sistema es disperso.
- Las mallas computacionales son generalmente irregulares, por lo que la matriz de las ecuaciones lineales no sólo es dispersa, sino también no estructurada.

A partir de esta descripción, es evidente que los solucionadores lineales son el primer candidato para la aceleración de la *GPU* en los programas de *CFD*. Por lo tanto, suponemos que la solución del campo de flujo y de presiones en el aspirador de la Turbina 99 será eficiente cuando un *GPU* resuelva el sistema de ecuaciones lineales obtenidas de la discretización de las ecuaciones de Navier-Stokes de la forma general:

$$Ax = b$$

Donde A es una matriz no estructurada, dispersa y real; x, son vectores reales. Para resolver este sistema matricial se han desarrollado un gran número de técnicas especiales. Entre estas técnicas, se tienen los métodos iterativos de Krylov que se han convertido en un estándar para la resolución de los grandes problemas lineales. Se puede encontrar una revisión exhaustiva de estos métodos en [56] y detalles técnicos de sus implementaciones informáticas en [57]. El presente trabajo acelera dos métodos populares de Krylov : Gradiente Conjugado (*CG*) y bi gradiente conjugado estabilizado (*BiCGStab*). En los apéndices en las Figuras B8 y B9 se presenta el código implementado.

3.2. Justificación

A lo largo de los años la caja de Herramientas del *Software CFD*, han ido en aumento no solo en tamaño si no en complejidad. Esto debido a la misma complejidad de los fenómenos de transporte presentes en la Ingeniería. En los fluidos también se hace notar esta complejidad con el desarrollo de nuevas herramientas para la infinidad de problemas que se presentan. En los últimos años el interés industrial, energético y académico por las importantes pérdidas energéticas en la generación hidroeléctrica ha impulsado la necesidad de tener resultados de manera eficiente y rápida en el estudio del tubo de aspiración [4]. Debido a ello los tres Talleres

[6, 7, 8] han generado conocimientos acerca de este dispositivo por medio del desarrollo del flujo del fluido con la ayuda de herramientas numéricas complejas como la *CFD*.

Sin embargo para usar esta herramienta numérica de manera correcta, se debe de ser capaz de establecer condiciones de frontera y de flujo realmente válidas. Los números de Reynolds en estos dispositivos son siempre muy altos, por lo tanto la resolución de la malla debe de ser lo suficientemente fina donde existen los mayores gradientes del flujo. Además cerca de los alabes del rodete puede existir cavitación y los métodos numéricos para resolver estos efectos requieren un mayor número de celdas y menores incrementos de tiempo.

Una opción para lograr reducir las idealizaciones del flujo de fluidos en este dispositivo es evitar o reducir la utilización de los modelos de turbulencia, es decir que la gran variedad de las escalas de los vórtices sean capturados a través de una simulación directa con los métodos *Direct Numerical Simulation (DNS) y Large Eddy Simulation (LES)* que son los más costosos en tiempo computacional y que contienen una cantidad considerable de cálculos que son enviados al *hardware* al mismo tiempo bloqueando la memoria provocando un cuello de botella [3].

La herramienta de *CFD OpenFOAM* ha probado ser una excelente plataforma para cálculos de flujo de alta calidad en turbinas hidráulicas. Lo que hace interesante al software *OpenFOAM*® no solo es el paralelismo de sus algoritmos de cálculo en *CPU's* convencionales sino también la posibilidad de implementar los cálculos matriciales en un procesador gráfico (*GPU*). Esto es debido a su disponibilidad de adecuar el código de programación y particularmente el acceso a los algoritmos de solución que se usan para el cálculo del tubo de aspiración del proyecto T-99. Una desventaja notable en el uso de *software* comercial es que no se conoce el código fuente, en otras palabras no se tiene acceso al *API* o la interfaz encargada del cálculo matricial. Así, los resolvedores o bien los algoritmos a resolver por el hardware es en donde cabe la posibilidad de canalizar órdenes por medio de *software* para mandar los cálculos en paralelismo fino a la arquitectura del *hardware GPGPU* .

3.3. Hipótesis

La Dinámica de Fluidos Computacional finalmente pertenece al área donde la arquitectura de un *GPU* puede mostrar sus ventajas ya que como los juegos de datos usados en los problemas de simulación de flujo son muy grandes, se pueden separar de manera relativamente fácil en pequeños grupos que a su vez pueden ser procesados en paralelo por hilos de ejecución (*threads*) individuales.

De esta forma, bajo la premisa de que los *GPU's* pueden acelerar cálculos en un sistema de ecuaciones lineales, el tener acceso a los diferentes códigos del *software CFD OpenFOAM* permitirá su implementación para lograr paralelizar masivamente en una simulación en 3D que discretize las ecuaciones de Navier-Stokes como es el caso de estudio de la Turbina-99 obteniendo así una potencia de cálculo mayor que la obtenida por procesadores convencionales multi-núcleo .

3.4. Objetivos

3.4.1. Objetivo general

Acelerar el tiempo de cálculo de la simulación del tubo de aspiración del proyecto Turbina-99 a través de la implementación del cálculo de alto rendimiento para un uso de memoria paralelizada del procesador gráfico de propósito general (*GPGPU*) adecuando librerías de cálculos matriciales en *C++* y usando a *OpenFOAM* como simulador computacional de fluidos.

3.4.2. Objetivos específicos

1. Implementar la memoria paralelizada del dispositivo de propósito General *GPGPU*.
2. Instalación del OpenFOAM® en una plataforma Linux y modificar el *toolbox*, para el uso de paralelismo fino.
3. Modelar numéricamente un problema de interés industrial; Tubo de Aspirador de una Turbina-99 tipo Kaplan.
4. Asegurar una alta calidad en las soluciones proporcionadas por el código *CFD* al cuantificar la incertidumbre en los resultados obtenidos.
5. Medir los parámetros involucrados en la aceleración de cálculos de procesamiento a través de pruebas de alto rendimiento computacional.

3.5. Metodología

Para establecer de manera específica, y con el fin de dar una estructuración al presente trabajo se lleva a cabo una metodología. De manera gráfica la Tabla 3.1 muestra la metodo-

logía desarrollada para el presente trabajo. De manera detallada se presentan los puntos de la metodología planteada de la manera siguiente:

1. Establecer las condiciones del problema teóricamente: esto involucra las condiciones de frontera que se tienen que implementar en el código *OpenFOAM* ,condiciones como en qué estado se realizará el problema.

2. Seleccionar la geometría, mallado y leer la malla en *OpenFOAM*. En este punto se establece la geometría, el mallado y variables donde se localizan los puntos de medición críticos.

3. Establecer en los scripts de *OpenFOAM* las condiciones de frontera. Esta tarea es meramente computacional donde se establece la configuración del problema y de acuerdo a la teoría presentada anteriormente se establece el directorio *Constant*.

4. Establecer los algoritmos de resolución y el control de la solución. Esta tarea es meramente computacional donde se establece los algoritmos que se usan para resolver el problema del tubo de aspiración, de acuerdo a la teoría presentada en el capitulo 1 se establece el directorio *System*.

5. Establecer en los scripts de *OpenFOAM* las condiciones de iniciales. De acuerdo a la teoría presentada anteriormente se establece el directorio 0.

6. Acoplar el *GPGPU* por medio de adaptación del código con software de terceros. El acoplamiento que se realiza es por medio de librerías externas, librerías mencionadas teóricamente en el capítulo anterior y sobretodo con el sistema operativo *GNU/Linux Ubuntu*.

7. Instalar *CUDA* y librerías necesarias para el cálculo en el GPU.

8. Acoplar por medio de *cufflink-library* a *OpenFOAM*.

9. Simular el problema en paralelo con *multi-CPU* y con la combinación de *GPU + CPU*. Esta tarea consiste en ejecutar los casos establecidos con las condiciones de frontera e iniciales y configurarlas.

10. Validar los resultados obtenidos en *multi-CPU*, *single CPU + GPU y GPU + multi-CPU*. En este punto se comparan cualitativa y quantitativamente los resultados numéricos contra los resultados obtenidos experimentalmente de mediciones y variables críticas estudiadas en la bibliografía.

Tabla 3.1: Metodología planteada para llevar a cabo el proyecto

11. Realizar las pruebas de cómputo de alto rendimiento y análisis de resultados. Aquí se comparan parámetros que involucran la aceleración del cálculo y donde también se miden cuantitativamente los tiempos de cálculo computacional.

Capítulo 4

Validación de resultados del tubo de aspiración de la turbina T-99

Turbina T-99 es uno de los proyectos más grandes internacionalmente, estos han sido dedicados a la investigación del tubo de aspiración tipo *sharp-heel* de una turbina tipo Kaplan U9 ubicada en Porjus, Suecia. El Proyecto apunta a validar simulaciones CFD donde diferentes grupos de investigación se reunieron para una discusión y divulgación de los resultados obtenidos. La geometría, 4.1 así como la malla, Fig. 4.4, se obtuvo por medio del contacto personal con el encargado del tercer taller internacional el Dr. M.J. Cervantes[8].

4.1. Características del modelo

El modelo del tubo de aspiración de la T-99 ha sido estudiado en tres talleres desarrollados durante la pasada década. Los datos del modelo se presentan a continuación:

- Salto: $H = 4.5\ m$
- Velocidad del rodete: $N = 595\ RPM$
- Flujo: $Q = 0.522\ m^3\ /s$
- Velocidad unitaria del rodete: $DN/\sqrt{H} = 140$, donde D es el Diámetro del rodete.
- Flujo unitario: $Q/D^2\sqrt{H} = 1.00$

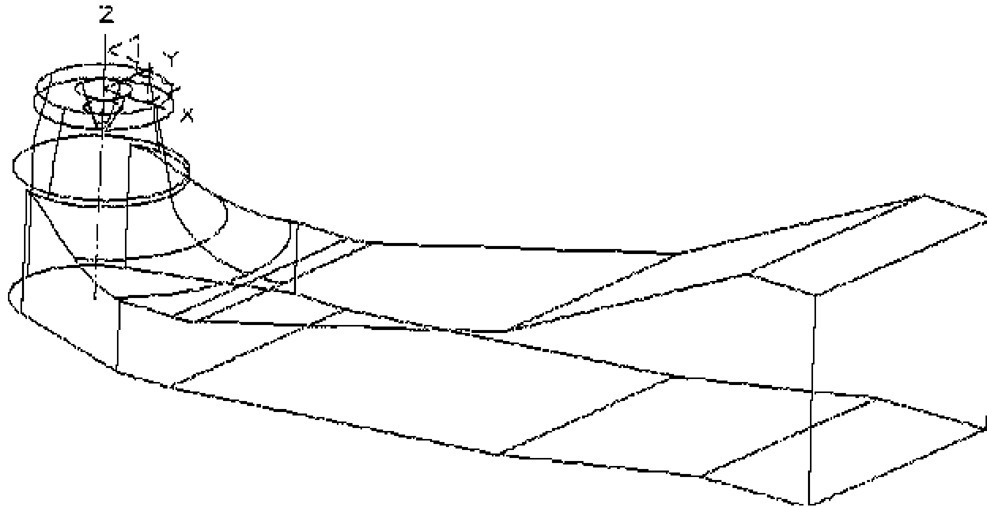

Figura 4.1: Esquema del tubo de aspiración de la turbina T-99 de [7]

- Temperatura del fluido: *15 ℃*

4.2. Condiciones de Frontera

Para establecer las condiciones de frontera en la entrada, las mediciones de velocidad fueron realizadas a lo largo de una línea radial a CS_Ia, Fig. 4.2 y 4.3 (*CS* representa la sección transversal). Se asumió que las componentes de velocidad y cantidades turbulentas no eran funciones de la posición azimutal, es decir es axisimétrico. La velocidad media axial (U), la velocidad media tangencial (W) y las cantidades turbulentas ($\bar{u^2}, \bar{v^2}, \bar{uw}$) se presentan en la tabla 4.1 ; donde los valores denotados con "*" estan situados en la pared y "**" son valores extrapolados de los experimentales. Sin embargo, ya que el conjunto de datos experimentales no está completo para llevar a cabo la simulación, se establecen suposiciones en la velocidad radial (V) y en las cantidades del flujo turbulento generado por el rodete (v^2, uv, vw) .

IREQ (Hydro-Québec's Research Institute), Instituto que ha dedicado en parte a la investigación de la *Turbina-99* desarrollando diversos trabajos de investigación ha proporcionado

Tabla 4.1: Velocidades que fueron establecidas como condiciones de frontera en la entrada o bien en la sección CS_Ia de [7].

Radius [m]	U [m/s]	W [m/s]	u [m/s]	w [m/s]	uw** [m^2/s^2]
0.0981*	0	6.11	0	0	0
0.1014	2.44	0.37	0.63	0.49	-0.011
0.1041	2.55	0.58	0.70	0.51	-0.020
0.1068	2.72	0.72	0.72	0.61	-0.030
0.1094	2.86	0.69	0.68	0.48	-0.038
0.1161	3.07	0.65	0.51	0.39	-0.060
0.1227	3.06	0.61	0.50	0.36	-0.050
0.1294	3.14	0.60	0.32	0.37	-0.030
0.1427	3.27	0.65	0.30	0.37	-0.020
0.1560	3.38	0.79	0.28	0.38	-0.030
0.1693	3.41	0.91	0.30	0.37	-0.040
0.1826	3.57	1.07	0.38	0.47	-0.048
0.1959	3.69	1.31	0.42	0.70	-0.121
0.2092	3.71	1.28	0.34	0.52	-0.047
0.2168	3.65	1.29	0.47	0.49	-0.021
0.2221	3.55	1.43	0.54	0.63	-0.002
0.2261	3.55	1.57	0.51	0.80	0.015
0.2301	3.54	1.59	0.57	0.83	0.033
0.2314	3.48	1.56	0.73	0.79	0.038
0.2328	3.40	1.52	0.79	0.73	0.036
0.2341	3.16	1.45	0.98	0.65	0.031
0.2365*	0	0	0	0	0

información al presente trabajo de las condiciones de Frontera en la entrada. Debido a ello se logró obtener con éxito la condición de entrada como perfil de velocidad, usando una librería dinámica en *C++(libOpenFoamTurbo.S)*, librería dedicada a turbo maquinaria para la condición inicial de la velocidad U, en el directorio de la estructura descrita con anterioridad *$case...0/U* (siendo $case la variable donde se encuentran todos los directorios necesarios para la simulación [38].

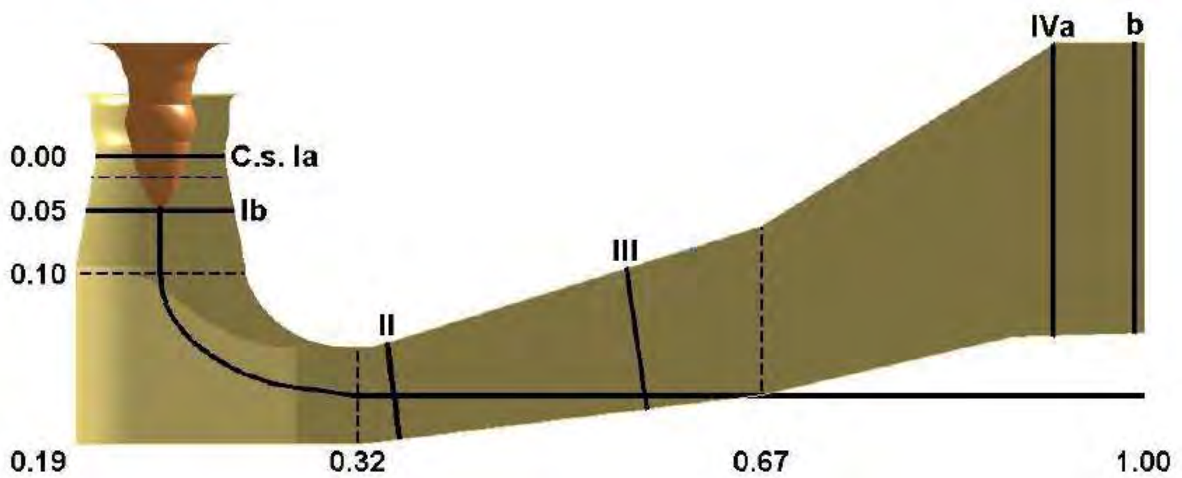

Figura 4.2: Secciones de medición planteadas para el T-99 de [7]

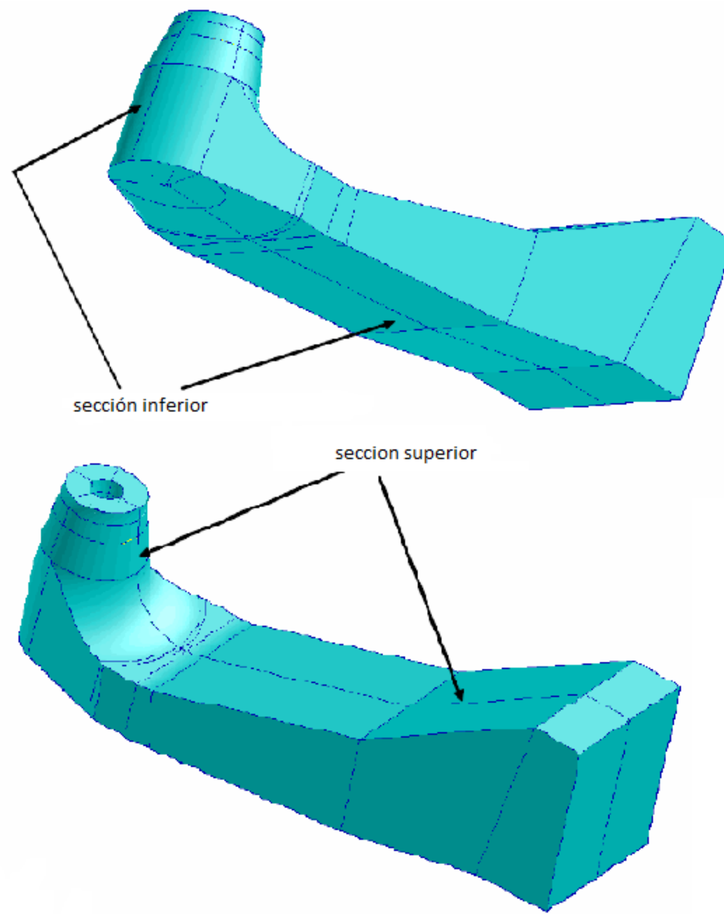

Figura 4.3: Línea de medición a lo largo de la pared superior e inferior del tubo de aspiración[8].

4.2.1. Modelo matemático de turbulencia $\kappa - \varepsilon$ estándar

Una forma de modelar la turbulencia es aplicar el método *RANS* (*Reynolds Average Navier-Stokes*), el cual consiste en sustituir las velocidades de las ecuaciones de Navier-Stokes por una velocidad instantánea, la cual es la suma de la velocidad media en el tiempo y la componente de velocidad fluctuante, e integrarlas sobre un intervalo de tiempo mayor que el tiempo característico de la turbulencia [31]. Bajo esta condición la ecuación de Navier Stokes es:

$$\frac{\partial \rho \bar{u}_i}{\partial x_i} = 0 \tag{4.1}$$

$$\rho \bar{u}_j \frac{\partial \bar{u}_i}{\partial x_j} = \frac{\partial P}{\partial x_i} + \frac{\partial}{\partial x_j}\left[\mu \frac{\partial \bar{u}_i}{\partial x_j} - \rho \overline{u_i' u_j'}\right] + \rho g_i \quad (4.2)$$

Como se puede observar, ya no sólo se deben calcular las incógnitas de la ecuación de transporte, sino también, los términos extras o esfuerzos de Reynolds, surgiendo así dos tendencias que buscan dar solución a las ecuaciones del método *RANS*. El modelo $\kappa - \varepsilon$ es uno de los modelos de turbulencia más implantados a nivel industrial. Es un modelo con dos ecuaciones de transporte para representar las propiedades turbulentas del flujo. La primera variable de este modelo es la energía cinética turbulenta κ (*Kappa*), dicha variable determina la intensidad turbulenta, mientras que la segunda variable representa la disipación turbulenta ε (*Epsilon*). Las ecuaciones que gobiernan dichas variables son las siguientes:

$$\frac{\partial}{\partial t}(\rho\kappa) + \frac{\partial}{\partial x_i}(\rho\kappa u_i) = \frac{\partial}{\partial x_j}\left[\left(\mu + \frac{\mu_t}{\sigma_k}\right)\frac{\partial \kappa}{\partial x_j}\right] + G_k + G_b - \rho_\epsilon - Y_M + S_\kappa \quad (4.3)$$

$$\frac{\partial}{\partial t}(\rho\epsilon) + \frac{\partial}{\partial x_i}(\rho\epsilon u_i) = \frac{\partial}{\partial x_j}\left[\left(\mu + \frac{\mu_t}{\sigma_\epsilon}\right)\frac{\partial \epsilon}{\partial x_j}\right] + C_{1\epsilon} + \frac{\epsilon}{\kappa}\left(G_\kappa + C_{3\epsilon} G_b\right) - C_{2\epsilon}\rho\frac{\epsilon^2}{\kappa} + S\epsilon \quad (4.4)$$

G_k:Generación de energía cinética turbulenta debido a los gradientes de velocidad medios.
G_b:Generación de energía cinética debido a la flotabilidad.
Y_M: Contribución de la dilatación fluctuante en turbulencia compresible.
$C_{1\epsilon}$, $C_{2\epsilon}$, $C_{3\epsilon}$, σ_t: Constantes determinadas experimentalmente.
μ:Viscosidad turbulenta.
σ_k:Número de Prandtl en función de k.
σ_ϵ: Número de Prandtl en función de ϵ.

Estas dos tendencias consisten en resolver los esfuerzos de Reynolds por medio de ecuaciones de transporte extras y/o por la aproximación de Boussinesq y los modelos de turbulencia que a partir de ella han sido creados.

El modelo κ-épsilon es uno de los más comunes modelos de turbulencia , a pesar de que no funciona bien en los casos de grandes gradientes de presión adversos [34]. Se trata de un modelo de dos ecuaciones, es decir, incluye dos ecuaciones de transporte adicionales para representar las propiedades de turbulencia del flujo. Esto permite un modelo de dos ecuaciones para explicar los efectos fluidodinámicos como la convección y la difusión de la energía turbulenta. La primera variable de transporte es la energía cinética turbulenta κ y la segunda variable de transporte es la disipación turbulenta ε, esta es la variable que determina la escala de la turbulencia, mientras que la primera variable, determina la energía

en la turbulencia. Hay dos formulaciones principales de modelos de $\kappa - \varepsilon$ [33] que Launder y Sharma típicamente lo llaman “el modelo de turbulencia estándar”.

El ímpetu original para el modelo $\kappa - \varepsilon$ estandar era mejorar el modelo de longitud de mezclado, así como encontrar una alternativa a la prescripción algebraicamente de las escalas de longitud de turbulencia en los flujos de moderada y alta complejidad. Como se describe en la Referencia [34], el modelo $\kappa - \varepsilon$ ha demostrado ser útil para los flujos de la capa límite de deslizamiento con gradientes de presión relativamente pequeños. Del mismo modo, para los flujos delimitados por pared y flujos internos, el modelo da buenos resultados sólo en donde los casos en que los gradientes medios de presión son pequeños; exactitud que se ha demostrado disminuir experimentalmente cuando flujos contienen grandes gradientes de presión adversos. Se podría inferir por tanto que el modelo $\kappa - \varepsilon$ estandar es una opción apropiada para los problemas como entradas de flujos y compresores, aunque siempre se ha usado como una aproximación debido a que es un modelo robusto en cuanto a tiempo computacional y estabilidad de convergencia.

4.3. Mallas del Problema T-99

Se sabe por antecedentes que un método numérico alcanza su máxima cercanía a lo experimental cuando el problema en estudio es bien discretizado, esto quiere decir que para el modelado numérico en CFD y la confiabilidad de sus resultados se necesita de una malla muy densa, es por ello que el presente estudio hace uso de éstas con un número de elementos y estructurada aproximadamente de:

- 1 millón de elemento
- 2 millones de elementos
- 6 millones de elementos

Las mallas para las pruebas de rendimiento (Figura 4.4) fueron proporcionadas por la Universidad de *LULEA* por medio de contacto personal, éstas han sido estudiadas en los tres *Workshops* T-99 dedicados al estudio del tubo de aspiración [8]. La convergencia de estas mallas, así como los métodos de mallado se han estado revisando por la Universidad de *LULEA* y la información proporcionada es la última.

Las mallas se trasladan desde un software comercial a *OpenFOAM* por medio de aplicaciones como es *cfxToFoam*, *Fluent3DToFoam*, etc,. Se ha checado la calidad de la malla

antes y después del traslado de *OpenFOAM* por medio de la aplicación de *checkMesh* de *OpenFOAM*.

El resultado del traslado de la malla ha sido una serie de puntos que *OpenFOAM* reconoce y que se encuentra en nuestra estructura descrita anteriormente en el directorio del caso en *$FOAM_ USER... constant/polymesh*.

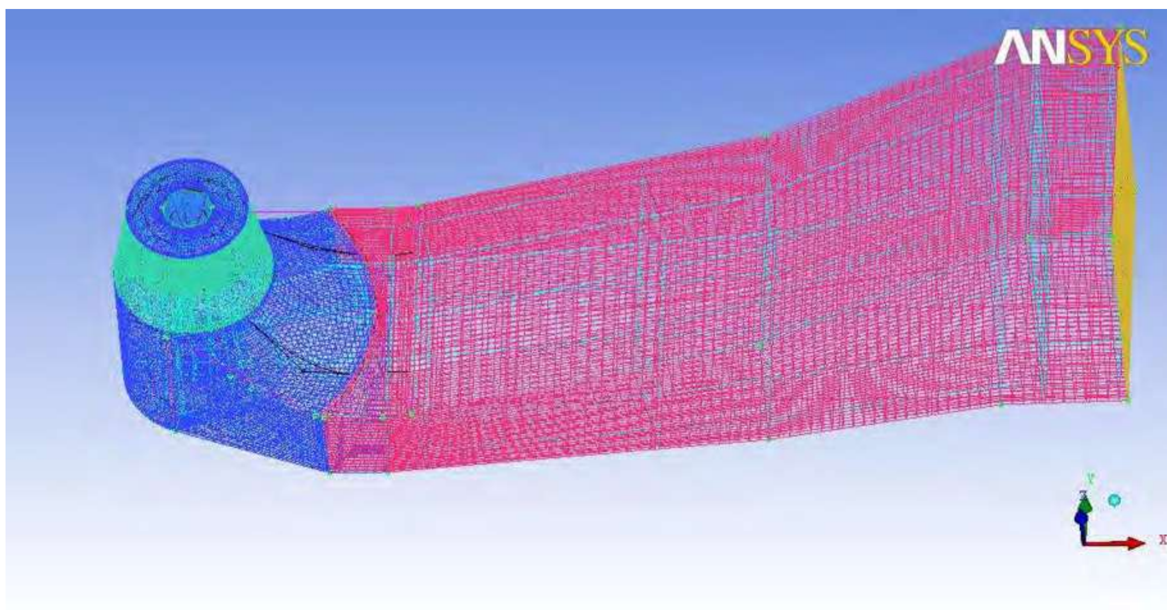

Figura 4.4: Mallado de las paredes en ICEM de ANSYS, cortesia la Universidad de LULEA

Para la validación sólo se hace uso de la malla de 1 millón de elementos y para el estudio computacional en el capítulo 5 se usan los tres diferentes mallados, esto es debido a que por antecedentes se sabe que la cantidad de trabajo computacional que se manda al *hardware* afecta directamente a la eficiencia computacional de la resolución de los algoritmos de cálculo. La manera en que se discretiza el dominio es por el metodo SIMPLE de *OpenFOAM*, donde se estará dividiendo el dominio por medio de su geometria en los tres ejes.

4.4. Resultados en las secciones de Medición

Se realizan pruebas en el simulador por medio del *software OpenFOAM* y se comparan resultados en las secciones inferior y superior (Figura 4.5) por medio de la ecuación de C_{pr} o coeficiente de recuperación de presión, el cual se obtiene con la medición de los puntos de presión a lo largo de la línea de la parte inferior y superior, el cual es comparado contra la presión de entrada, de manera matemática se tiene:

$$C_{pr} = \frac{P_{out:wall} - P_{in:wall}}{P_{dyn,Ia}} \tag{4.5}$$

Se observa que para realizar el análisis comparativo de los datos experimentales de la recuperación de presión C_{pr} en el tubo de aspiración T-99 se necesitan 3 cantidades ingenieriles , $P_{out:wall}$ es la presión estática promedio a lo largo de la sección IVb (4.2); $P_{in:wall}$ es la presión estática promedio en la sección *Ia* y finalmente $P_{dyn,Ia}$ es la presión dinámica en la sección Ia y por medio de la cual es calculada de la siguiente forma:

$$P_{dyn,Ia} = \frac{\rho Q^2}{2A_{Ia}^2} \tag{4.6}$$

Los resultados arrojados para el factor de recuperación de presión se dividen en 2 secciones (Figura 4.3) de acuerdo al proyecto T-99 la sección inferior y la superior, donde en la Figura 4.5 se muestran las comparaciones graficas de manera adimensional a lo largo de la pared para la sección superior e inferior respectivamente. Ambas muestran la comparación de los resultados entre los medidos experimentalmente y los realizados en este proyecto con el simulador de fluidos *OpenFOAM* donde además sus cálculos matriciales fueron hechos en la unidad *GPGPU*.

Los contornos de medición de la velocidad que se comparan experimentalmente contra los resultados del presente trabajo son para la malla de 1 millón de elementos y se tienen en cuenta aspectos como es la velocidad normal al plano en las sección II y en el plano en la sección III para medir su velocidad.

En la Fig. 4.5, en general, existe una buena aproximación de los valores del Cpr obtenidos tanto en la pared superior como en la inferior. En términos de una representación cualitativa de los resultados, las Figura 4.8 proporciona una visualización en forma de contornos de la velocidad normal a la sección Cs2. Aquí se puede ver una importante diferencia contra los valores experimentales de los resultados de la simulación, tanto de *FLUENT* como de *OpenFOAM*. Sin embargo, la relación entre resultados obtenidos entre los software es muy parecida. Y eso es el resultado de la utilización del modelo k-e estándar, quien en este caso

no captura la zona de velocidad en la parte media de la sección. Esto mismo sucede para la velocidad horizontal en la seccion Cs3 y que se muestra en la Fig. 4.9.

Por último las líneas de corriente desde el cono de expansión donde el fenómeno característico del tubo de aspiración llamado "torcha" está presente y las secciones de medición para la variable velocidad excepto la sección I se encuentran en la Fig. 4.6, resultados de acuerdo a la Fig. 4.2 arrojados por *OpenFOAM* y post-procesado en *ANSYS CFD Post* . También está representada la velocidad W cerca del cono de expansión en la Figura 4.7 como se realiza en [26] .

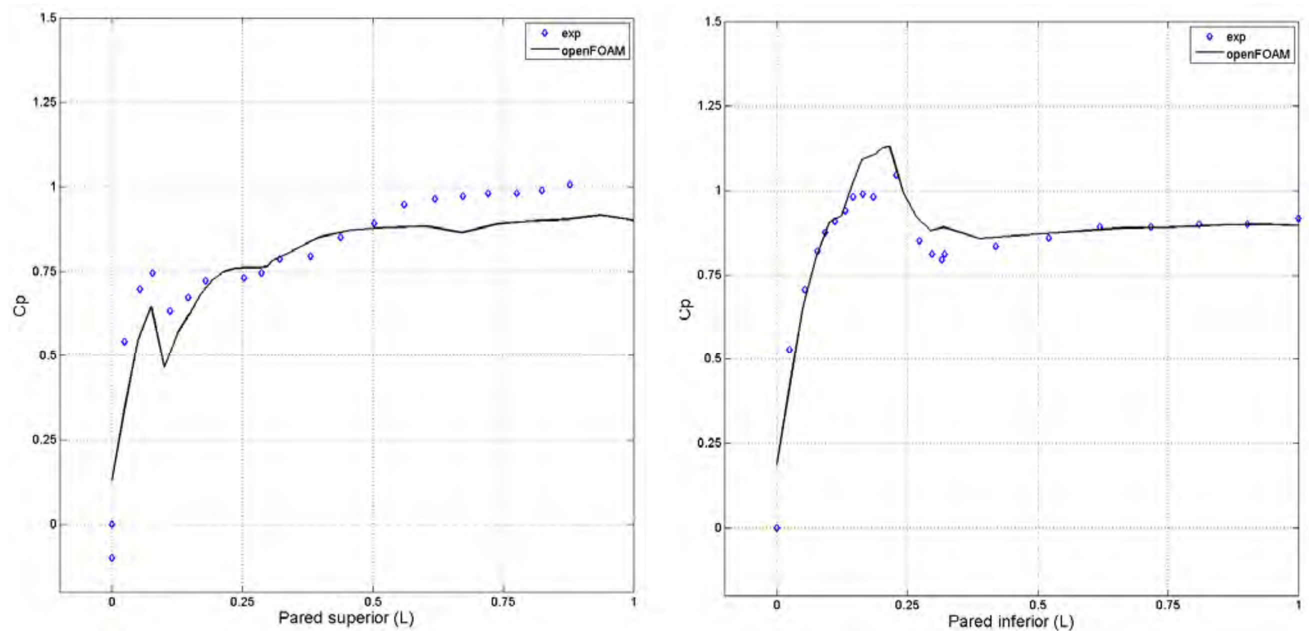

Figura 4.5: Grafica comparativa de Cpr en la sección superior e inferior respectivamente.

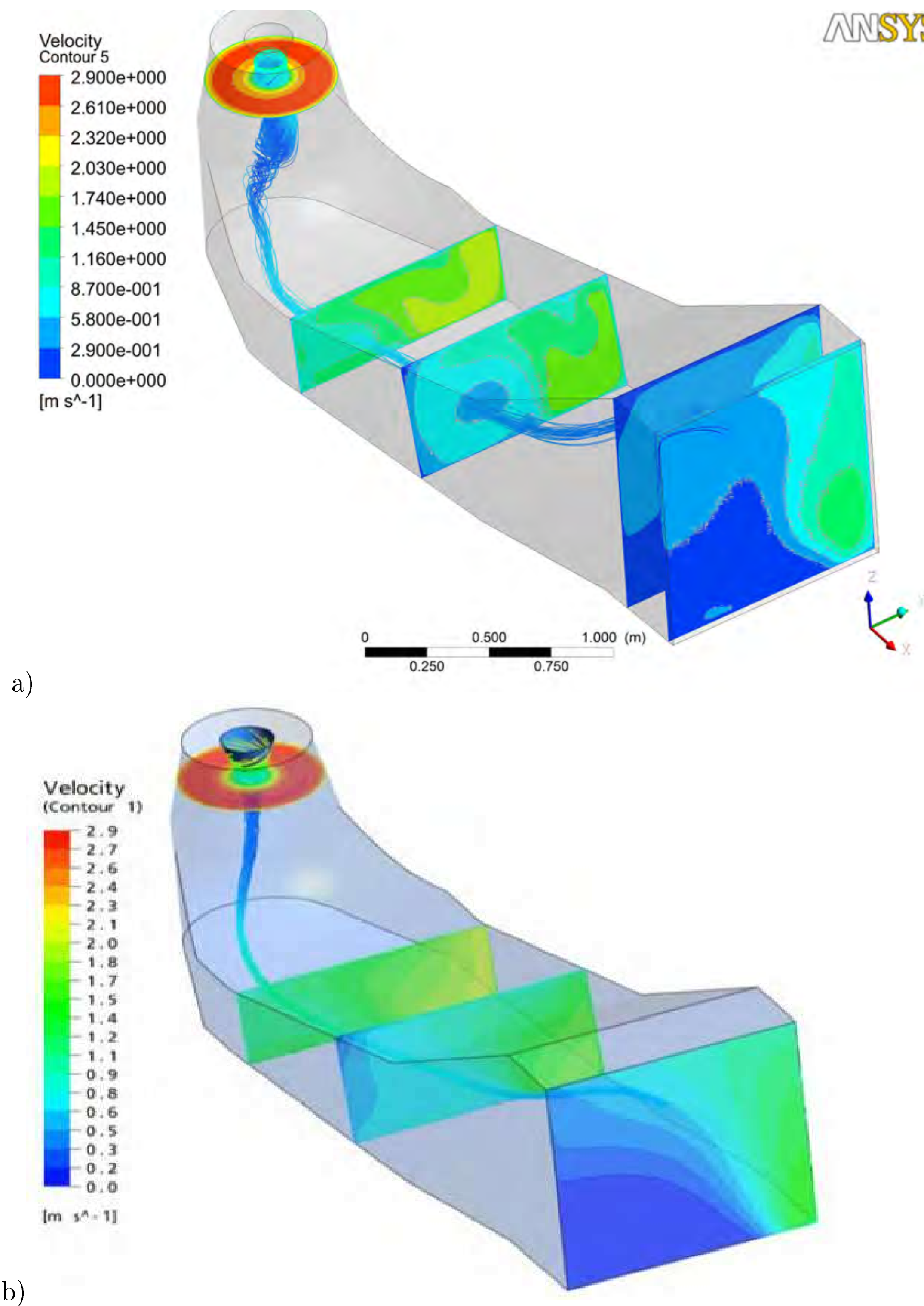

Figura 4.6: a) Líneas de corriente desde el cono de expansión y contornos de velocidad para las secciones Ib, II, III, IV y IVb arrojados por OpenFOAM (post-procesamiento en ANSYS©) en comparación al trabajo b) Líneas de corriente desde el cono de expansión y contornos de velocidad para las secciones por Ib, II, III, y IVb por M.J. Cervantes, [26]. .

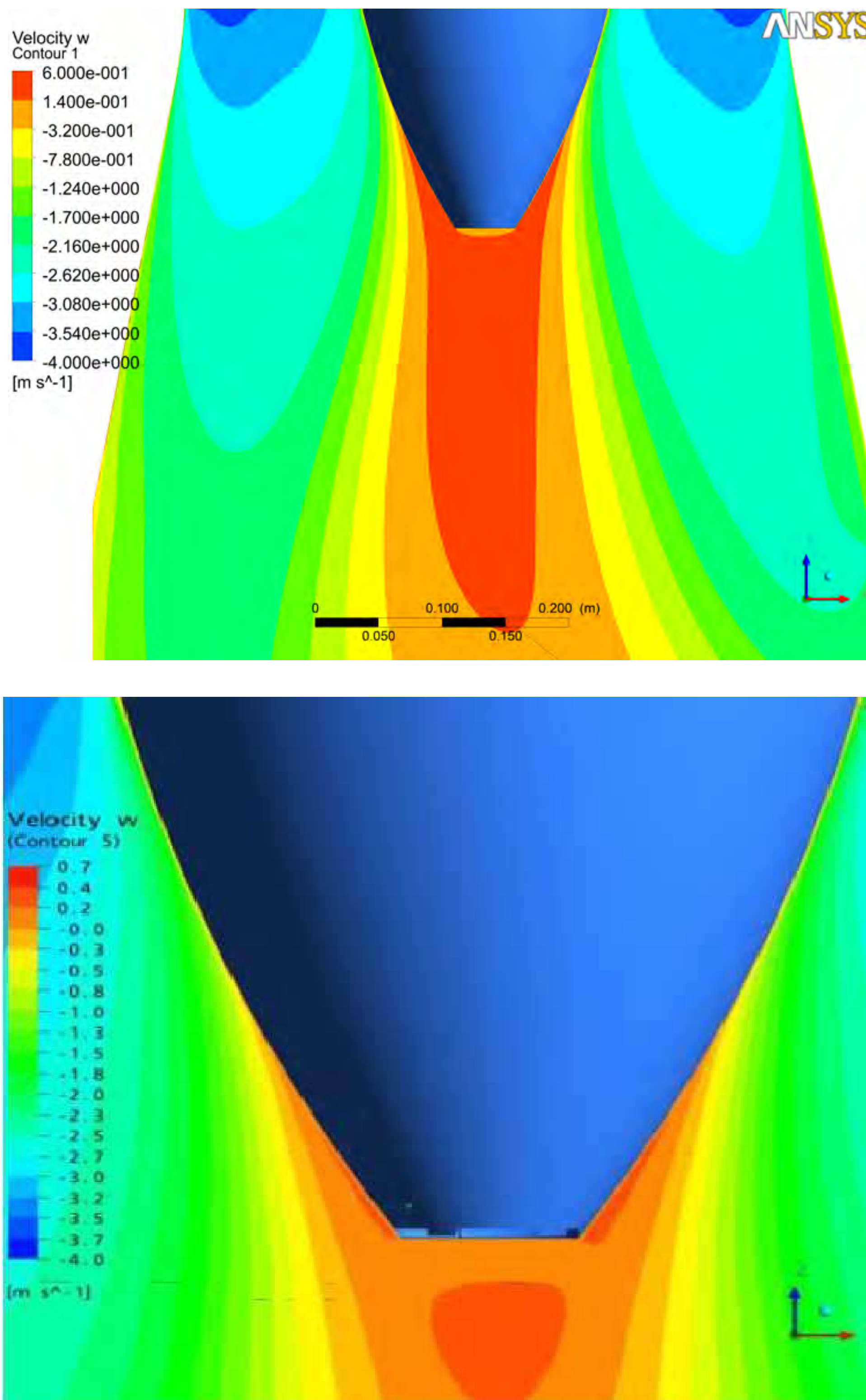

Figura 4.7: Contornos de velocidad cerca del cono de expansión del presente trabajo en comparación al trabajo realizado por M.J. Cervantes, [26], respectivamente.

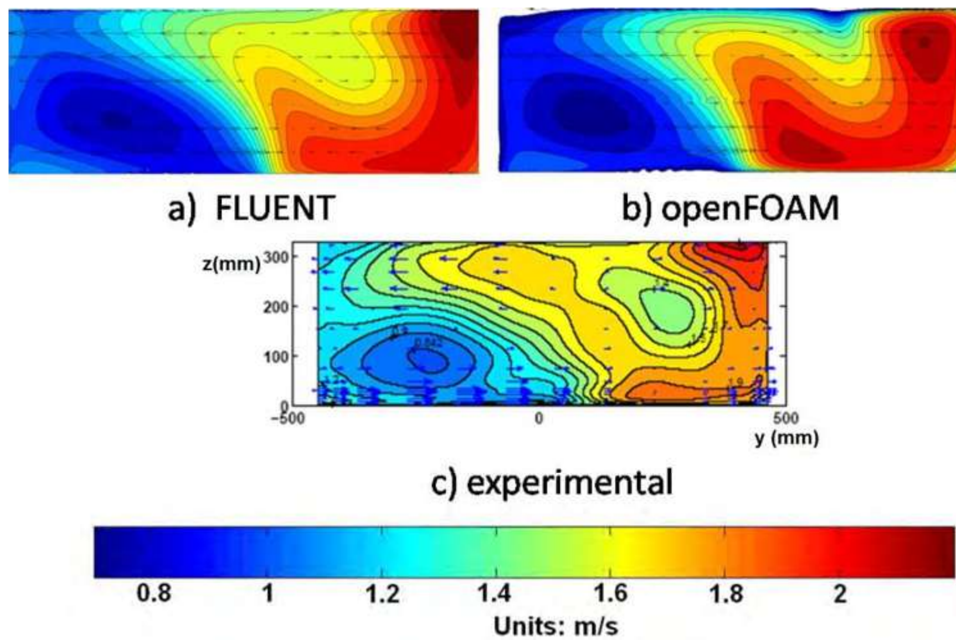

Figura 4.8: Contornos de velocidad normal a la sección II (CS_II).

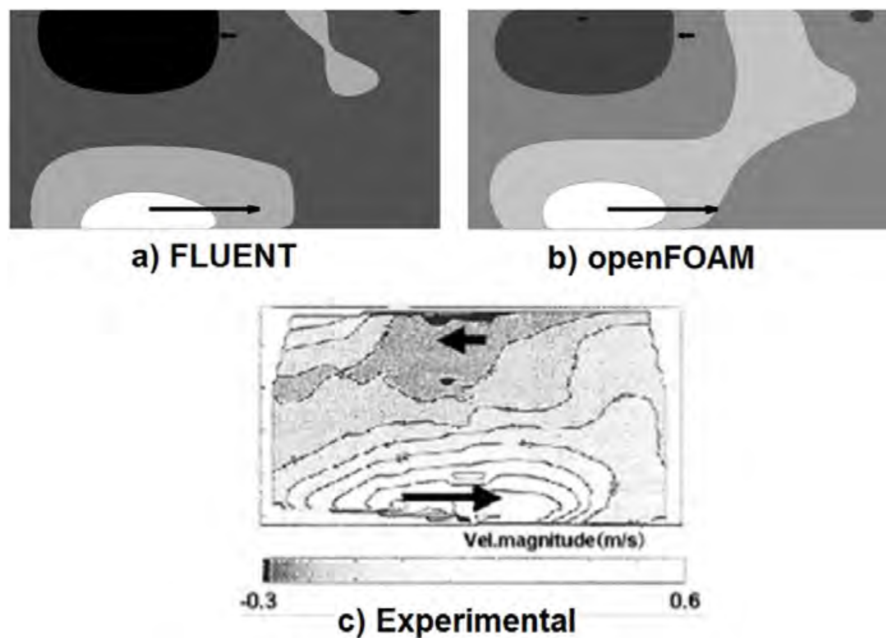

Figura 4.9: Contornos de velocidad horizontal a la sección III (CS_III).

Capítulo 5

Pruebas de *HPC* y resultados cuantitativos computacionales

En este capítulo se realiza un *benchmark* computacional, el cual sirve como un estándar para evaluar el rendimiento de un sistema o componente del mismo. La palabra *benchmark* es un anglicismo traducible al español como “comparativa”. Si bien también puede encontrarse esta palabra haciendo referencia al significado original en la lengua anglosajona, es en el campo informático donde su uso se extiende ampliamente.

Más formalmente puede entenderse que un *benchmark* es el resultado de la ejecución de un programa informático o un conjunto de programas en una máquina, con el objetivo de estimar el rendimiento de un elemento concreto, y poder comparar los resultados con máquinas similares.

En términos de ordenadores, un *benchmark* podría ser realizado en cualquiera de sus componentes, ya sea *CPU*, *RAM*, tarjeta gráfica, etc. También puede ser dirigido específicamente a una función dentro de un componente, por ejemplo, la unidad de coma flotante del *CPU*; o incluso a otros programas. La tarea de ejecutar un benchmark originalmente se reducía a estimar el tiempo de proceso que lleva la ejecución de un programa (medida por lo general en miles o millones de operaciones por segundo).

Con el paso del tiempo, la mejora en los compiladores y la gran variedad de arquitecturas y situaciones existentes convirtieron a esta técnica en toda una especialidad. La elección de las condiciones bajo la cual dos sistemas distintos pueden compararse entre sí es ardua epecialmente en la publicación de los resultados.

Es por ello que el presente trabajo hace un estandarizado del modelo numérico del tubo de aspiración de la Turbina-99 y hace contraste en cuanto a la cantidad de cálculo computacional

mandado al *hardware*, ya que se aumenta la cantidad de elementos en la malla. Este modelo numérico es probado con diferentes especificaciones que son establecidas gradualmente, aumentando su potencia en una sola estación.

5.1. Equipo de cálculo

La arquitectura *CUDA* fue implementada por primera vez en la tarjeta gráfica serie G8x en 2007. Se ha estado mejorando continuamente de manera extraordinaria y es una de las últimas tecnologías y versión de *CUDA*. La arquitectura que se usa en este proyecto se llama "Fermi".

La arquitectura *CUDA Fermi* hizo su primera aparición en el mercado en 2010. Sin embargo Nvidia afirma que los programas desarrollados para la serie G8x también funcionarán sin modificaciones en todas las futuras tarjetas gráficas Nvidia, debido a la compatibilidad binaria.

La arquitectura *CUDA* ofrece varias ventajas:

- Descarga rápida de datos y lecturas rápidas desde y hacia el *GPU*.
- Soporte completo para operaciones de enteros y operaciones *bit* a *bit*.
- Memoria compartida rápida de hasta 48 kilobytes (kb) por varios procesadores.
- Lectura de direcciones arbitrarias de la memoria del *GPU*.

En el presente proyecto la arquitectura Fermi es puesta a prueba por medio del benchmark de la Turbina-99 haciendo uso de dos dispositivos *GPU*.

- Uno para la visualización y representación del fenómeno del flujo del tubo de aspiración.
- El otro como un co-procesador del *GPU* para la computación paralela de sus cálculos.

Tabla 5.1: Tabla de propiedades principales del equipo de trabajo

Características	MAQUINA *WSPAC*
Procesador	2 x Intel Xeon e5504 a 2.0 Ghz
Memoria RAM	12 Gb
Gpu gráficos	NVIDIA Quadro FX 3800
Gpu para cálculos	4 x NVIDIA TESLA C1060
Sistema operativo	Win7/ Ubuntu

Tabla 5.2: Especificaciones de la TESLA C1060

Formato	10,5" x 4,376", doble ranura
Nº de *GPU*'s Tesla	1
Nº de núcleos de procesamiento para streaming	240
Frecuencia de los núcleos de procesamiento	1.3 Ghz
Precisión de las operaciones en coma flotante	Precisión doble y simple según IEEE 754
Memoria total dedicada	4 GB
Frecuencia de la memoria	800 MHz
Interfaz de memoria/td>	512-bit GDDR3
Ancho de banda de memoria	102 GB/sec
Consumo máximo	187.8 W
Interfaz del sistema	PPCI Express x16 Generación 2
Conectores de alimentación auxiliares	dos de 6 patillas o uno de 8 patillas
Solución de disipación térmica	Ventilador/disipador activo
Entorno de programación	C(CUDA)

Como se explota totalmente la potencia de cálculo de los procesadores y de los dispositivos *GPU*, éstos podrían sobre-calentarse muy rápidamente, por lo tanto, las estaciones de trabajo y *PC*'s deben tener en su chasis una gran cantidad de ventilación incluyendo en el lado que están embonados con la placa base o placa madre. Es por ello que una estación de trabajo es una de las más adecuadas para realizar este trabajo computacional, siendo el indicado a usar por su chasis con espacio suficiente para adecuar la ventilación.

La tabla 5.1 enlista las propiedades principales de la estación de trabajo utilizada en este proyecto y que además influyen de manera importante en el rendimiento de cálculo. Ésta ya se encuentra adecuada correctamente con sus dispersores térmicos y dispositivos esenciales para una correcta funcionalidad, Fig. 5.1.

El *GPU Tesla C1060*, presentado en la Tabla 5.1, está basado en la serie de procesadores que son dotados de múltiples núcleos para cálculo masivo en paralelo, que se combina con el entorno de programación del *GPU*; C++ y *CUDA* a fin de simplificar la programación en sistemas multinúcleo.

Este *GPGPU*, Fig. 5.2, a pesar de ser uno de los primeros en el mercado, está dotado con 240 *CUDA cores* y la compatibilidad del lenguaje de programación con el *software OpenFOAM* es su principal ventaja.

Figura 5.1: Supercomputadora para el cálculo del benchmark de la T-99 .

Figura 5.2: Dispositivo *GPGPU*; *TESLA C1060*

5.2. Algoritmo de OpenFOAM a acelerar

En *OpenFOAM* es necesario establecer qué tipo de aplicación se va a usar y por antecedentes se va hacer uso del algoritmo *SIMPLE,* éste es ampliamente utilizado como procedimiento numérico en dinámica de fluidos computacional (*CFD*) para resolver las ecuaciones de Navier-Stokes. *SIMPLE* es un acrónimo para el Método Semi-implícito que se refiere a resolver el caso por medio de ecuaciones vinculadas a ecuaciones de presión.

El algoritmo *SIMPLE* fue desarrollado por el profesor Brian Spalding y por su estudiante Suhas Patankar en el *Imperial College* de Londres en la década de 1970. Desde entonces, ha sido ampliamente utilizado por muchos investigadores para resolver diferentes tipos de flujo de fluido y los problemas de transferencia de calor.

El algoritmo es iterativo. Los pasos básicos de solución son las siguientes:

1. Establecer las condiciones de contorno.

2. Cálculo de los gradientes de velocidad y presión. Las matrices resultantes de las velocidades y de las variables $\varkappa$ y ε son resueltas por medio del método Krylov; bi-gradiente conjugado estabilizado con un precondicionador diagonal. A diferencia de la presión se cambió de usar bi-gradiente conjugado estabilizado por el método: gradiente conjugado con un precondicionador de diagonal incompleta Cholesky.

3. Resolver la ecuación de momento discretizada para calcular el campo de velocidades intermedias.

4. Calcular los flujos de masa no corregidos en las caras.

5. Resolver la ecuación de corrección de presión para encontrar valores de las celdas de corrección de la presión, en este paso las matrices asimétricas resultantes son resueltas por medio del método de bi-gradiente congujado estabilizado con precondicionado diagonal.

6. Actualizar el campo de presión: $p^{k+1} = p^k + urf \bullet p'$ siendo urf el factor de debajo relajación de la presión.

7. Actualizar las correcciones de presión límite p'_b.

8. Corregir los flux másicos de las caras: $\dot{m}_f^{k+1} = \dot{m}_f^* + \dot{m}'_f$

9. Corregir las velocidades de los elementos: $\overrightarrow{v}^{k+1} = \overrightarrow{v}^{*} - \frac{Vol \nabla p'}{\overrightarrow{ap}^v}$; donde $\nabla p'$ es el gradiente de las correcciones de presión, $\overrightarrow{ap}^v$ es el vector de coeficientes centrales para el sistema lineal discretizado que representa la ecuación de velocidad y *Vol* es el volumen del elemento.

10. Actualizar la densidad debido a cambios de presión.

5.3. Resultados computacionales

Para poder llegar a estos resultados se presentaron algunos errores debido a que los programas informáticos paralelos son más difíciles de escribir que los secuenciales, porque la concurrencia introduce nuevos tipos de errores de *software*. Es por ello que la comunicación y sincronización entre diferentes subtareas son algunos de los mayores obstáculos que se tienen para obtener un buen rendimiento del programa paralelo y su solución se incluía en recompilar y depurar errores hasta llegar al resultado deseado.

Los presentes resultados muestran las comparativas entre realizar una de las tareas de *CFD* principales (resolución de matrices resultantes) desde usar una cantidad mínima de 2 núcleos del procesador central o *host* hasta llegar a la realización de los cálculos en el dispositivo (*GPU*) con un sólo hilo de ejecución que pasa de la memoria principal a la memoria del *GPU* la carga de matrices a resolver y también una ejecución donde se usan múltiples hilos de ejecución con acceso a la memoria de un solo *GPU*. Cabe mencionar que la máquina *WSPAC* tiene 4 unidades *GPU*, y que en este benchmark sólo se usara un solo *GPU*.

Se realizaron en total 18 ejecuciones de los cuales 6 casos fueron dedicados para cada una de las mallas propuestas en la sección anterior, en la tabla 5.3 se enlistan todos los casos para cada una de las mallas. El *benchmark* es para una cantidad de iteraciones fijas, los registros de salida tomados en cuenta es para una cantidad de 100 iteraciones.

Caso	*Hardware* de cálculo
2 cpu	Usa el *host* (*CPU*) discretizando en 2 dominios
4 cpu	Usa el *host* (*CPU*) discretizando en 4 dominios
6 cpu	Usa el *host* (*CPU*) discretizando en 6 dominios
8 cpu	Usa el *host* (*CPU*) discretizando en 8 dominios
gpu	Usa el *GPU* con 240 *CUDA cores* y 1 núcleo *CPU* realiza el "*envío*"
gpu+multi	Usa el *GPU* con 240 *CUDA cores* y 8 núcleos *CPU* realizan el "*envío*"

Tabla 5.3: Relación de casos para el benchmark

En la Tabla 5.3 se puede observar la palabra "*envío*", esta palabra se refiere al proceso que tiene que realizar el *CPU* para que el *GPU* realice los cálculos matriciales o bien el proceso *CUDA* (véase Fig. 2.3).

El acceso al *GPU* por medio de múltiples hilos de ejecución (caso "gpu+multi") es más rápido en la mayoría de los casos, sin embargo se observa en el presente trabajo la necesidad de la correcta discretización del dominio para el potente uso de combinación de *GPU* y *CPU*.

Es por ello que el presente trabajo se limita a realizar las ejecuciones en un *GPU*, a pesar de que la máquina *WSPAC* contiene 4 unidades disponibles. Y a cambio se compara el *GPU* entre sólo usar la cantidad mínima de núcleos disponibles y la máxima cantidad de núcleos disponibles para realizar el "*envio*" (*"gpu" y "gpu+multi"* respectivamente). El último caso ("*gpu+multi*") realiza la tarea con acceso a todos los procesadores de la máquina *WSPAC* (8 en total) usando la aplicación *Open MPI* (*Message Passing Interface*) que es el encargado de hacer las tareas en múltiples hilos de ejecución.

En el *benchmark* computacional que se llevó a cabo en el presente trabajo se homogenizaron los casos en cuanto a sus condiciones de frontera, cambios minimos para la convergencia de los casos también se han hecho y que además la configuración ha afectado un poco en el tiempo computacional (coeficientes de relajación). Se enlistan en las tablas 5.4, 5.5 y 5.6 los tiempos y los parámetros considerados para su comparación.

Se observa que existen dos tiempos que *OpenFOAM* toma en consideración y son registrados, uno de ellos es el tiempo de ejecución (*execution time*) que es el tiempo en segundos que toma en ejecutarse los cálculos que realiza *OpenFOAM* junto con el manejo de todos los *scripts* que hace el simulador de fluido. En contraste, el tiempo *wall-clock time* o tiempo real de procesamiento se puede observar en las gráficas 5.3,5.4 y 5.5 que es mayor debido ya que no solo registra el tiempo cálculo de los *scripts* realizados por *OpenFOAM*, sino que también los procesos coexistentes para que la computadora esté trabajando (por ejemplo; video, *SO*). Es importante registrarlo debido a que una gran diferencia entre estos tiempos registrados puede significar una disminución en la tarea principal de nuestro trabajo que es acelerar el cálculo.

Parámetros tomados en cuenta como es la aceleración, se calcula de la siguiente formula:

$$aceleración = \frac{razón1}{razónN} \tag{5.1}$$

Donde la $razón1$ es la razón de tiempo computacional del caso que tarde más tiempo en resolver o bien de dónde se referencía el *benchmark* y la $razónN$ es la razón de tiempo computacional del caso por nodos. Por ejemplo para el caso de "gpu+multi" basta con dividir la cantidad de 5077.42 entre 1738.55. Cabe mencionar que los nodos cuda y los nodos del procesador central no tienen la misma cantidad ni características como se puede observar en la tabla 5.2, su frecuencia es más baja que cualquiera de los procesadores comúnmente comerciales pero el *GPU* tiene en contraste 240 nodos disponibles (*cuda cores*) en total. El parámetro de la aceleración se comperá en contra de la cantidad de nodos disponibles y discretizados por el código *OpenFOAM*, por medio de discretización SIMPLE y se sabe que

esta relación es semi- linealmente computacional si el set-up es correcto como se observa en la figura 5.6.

Los valores de aceleración se encuentran en un valor arriba de la unidad en los casos más rápidos y que se puede observar en las tablas 5.4, 5.5 y 5.6, lo cual indica el comportamiento deseado. Esto significa que el trabajo computacional que realiza el *GPU* usando múltiples hilos de ejecución es 2.92 veces más rápido en la malla de 1 millón de elementos que solo realizarla con 2 núcleos del procesador central o *CPU*. También se puede observar que para el mismo mallado usando el *GPU* existe una ventaja considerable de 1.98 más rápido contra el uso del número máximo de núcleos disponibles en el *CPU* o bien ocho en total (véase tabla 5.4). Si se compara la máxima capacidad computacional del *CPU* contra la máxima capacidad computacional del *GPU*, el valor de aceleración sigue siendo arriba de la unidad para el caso del mallado de 1 millón de elementos, extrapolando se tiene un valor de 1.85 veces más rápido.

Caso	*execution time* (s)	*wall-clock time* (s)	aceleración	ahorro %	η
2 cpu	5077.42	5103	1.00000	192.04912	1.00000
4 cpu	4329.02	4336	1.07147	149.00175	0.426301
6 cpu	3767.65	3773	1.98971	116.71220	0.24734
8 cpu	3459.22	3477	2.16712	98.97155	0.17032
gpu	1862.81	3357	2.49001	7.14733	0.00135
gpu+multi	1738.55	1762	2.92049	0.00000	0.00126

Tabla 5.4: Tabla de tiempos computacionales para la malla de 1 millón de elementos

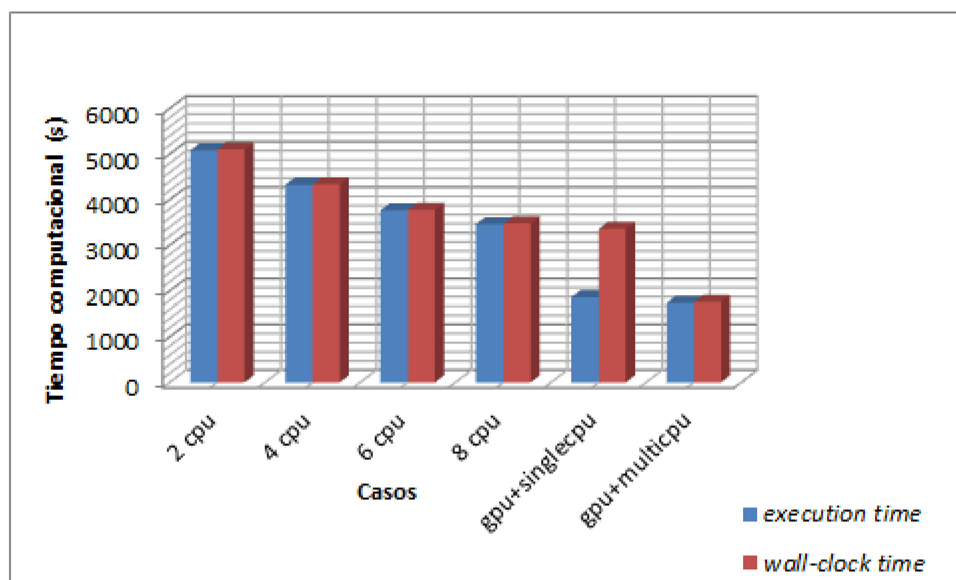

Figura 5.3: Gráfica de tiempos de cálculo para la malla de 1 millón de elementos

En la malla de 2 millones de elementos existe el mismo comportamiento de disminución del tiempo de cómputo, sin embargo existe una discrepancia en el uso de los dos casos planteados para el uso del *GPU*, donde el caso de "gpu" es aún más rápido que el caso de "gpu+multi" por 0.19 puntos unitarios de aceleración. Esto afirma que la discretización del dominio es crucial y no sólo para el cálculo matricial en el *CPU* sino para el manejo de datos de la memoria del *CPU* al *GPU* o proceso *CUDA*. Para este mallado se obtiene una aceleración de 2.5 veces más rápido usando el *GPU* contra el uso de 2 núcleos y 1.91 veces contra el uso de 8 núcleos del *CPU* (véase Tabla 5.5). Si se compara la máxima capacidad computacional del *CPU* contra la máxima capacidad computacional del *GPU* el valor de aceleración sigue siendo arriba de la unidad para el caso del mallado de 2 millones de elementos, extrapolando se tiene un valor de 1.91 veces más rápido.

Caso	*execution time* (s)	*wall-clock time* (s)	aceleración	ahorro %	η
2 cpu	11693.3	11719	1.00000	211.33067	1.0000
4 cpu	10891.9	10917	1.19381	189.99363	0.4657325
6 cpu	8604.89	8622	1.86446	129.10266	0.2452940
8 cpu	8838.31	9221	1.91504	135.31740	0.1889609
gpu	3755.91	3780	2.53364	0.00000	0.0011852
gpu+multi	4615.21	5559	2.36000	22.87860	0.0014564

Tabla 5.5: Tabla de tiempos computacionales para la malla de 2 millones de elementos

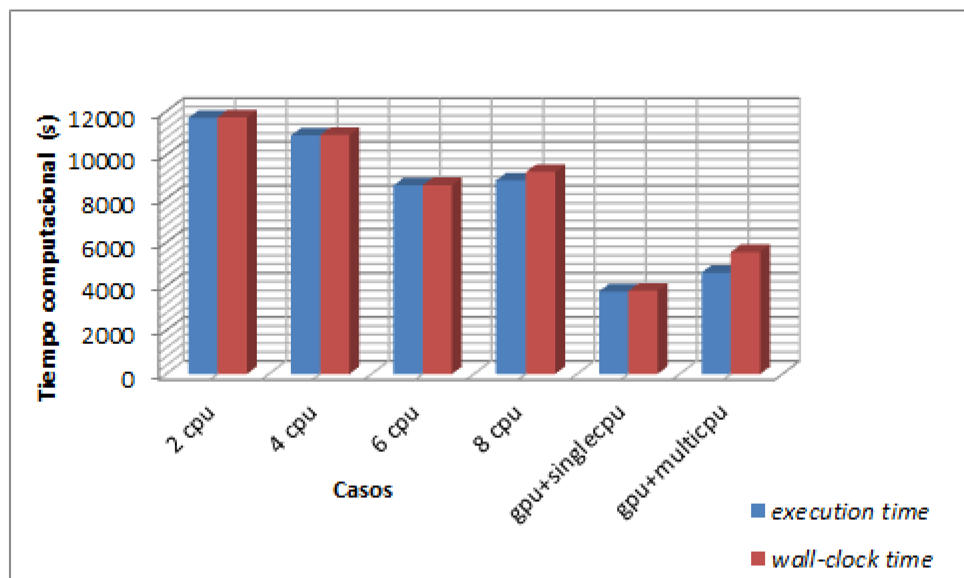

Figura 5.4: Gráfica de tiempos de cálculo para la malla de 2 millones de elementos

El comportamiento de la malla de 6 millones de elementos era de esperarse un aumento en los tiempos de computo pero existe un fenómeno interesante donde el valor de aceleración es aún mayor que en las mallas de 1 y 2 millones de elementos, esto nos indica que entre mayor sea la cantidad de trabajo computacional mayor será el valor de aceleración. Teniendo para este mallado el valor de la aceleración de 3.29 veces usando el *GPU* que solo usar dos núcleos del *CPU* y 2.48 veces más rápido contra el caso de ocho núcleos (véase Tabla 5.6). Cabe notar que para este mallado en el caso de "gpu+multi", la capacidad de los núcleos del *CPU* que estaban trabajando en el proceso *CUDA* se encontraban al 100 % de su capacidad, a contraste de la malla de 1 y 2 millones de elementos donde los núcleos estaban alrededor del 80 % de su capacidad. Si se compara la máxima capacidad computacional del *CPU* contra la máxima capacidad computacional del *GPU* el valor de aceleración sigue siendo arriba de la unidad para el caso del mallado de 6 millones de elementos, extrapolando se tiene un valor de 2.25 veces más rápido.

Caso	*execution time* (s)	*wall-clock time* (s)	aceleración	ahorro %	η
2 cpu	25369.7	25501	1.00000	212.30819	1.00000
4 cpu	26756.2	26805	1.11669	229.37639	0.52732
6 cpu	20218.4	20256	2.25967	148.89422	0.26565
8 cpu	18356	18853	2.48894	125.96755	0.18088
gpu	9071.21	9149	3.12308	11.66916	0.00131
gpu+multi	8123.29	8522	3.29376	0.00000	0.00118

Tabla 5.6: Tabla de tiempos computacionales para la malla de 6 millones de elementos

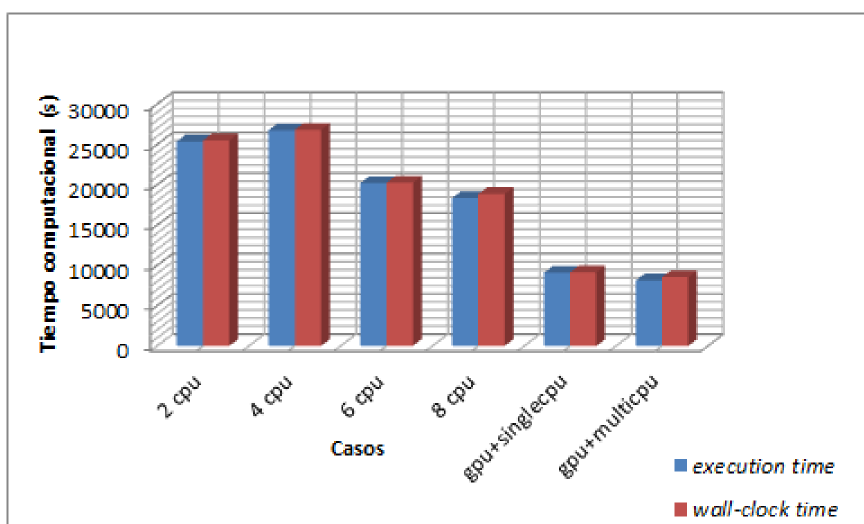

Figura 5.5: Gráfica de tiempos de cálculo para la malla de 6 millones de elementos

En la Fig. 5.6 se observa que la gráfica en el eje de los nodos se encuentra en escala logarítmica y es debido a la gran diferencia numérica entre los nodos de un *CPU* o *cores* disponibles y los nodos de un *GPU* o *CUDA* cores. También se puede apreciar cómo el valor tiende aumentar con el mayor uso de número de nodos con excepción de la malla de 2 millones de elementos que tiende a disminuir un poco.

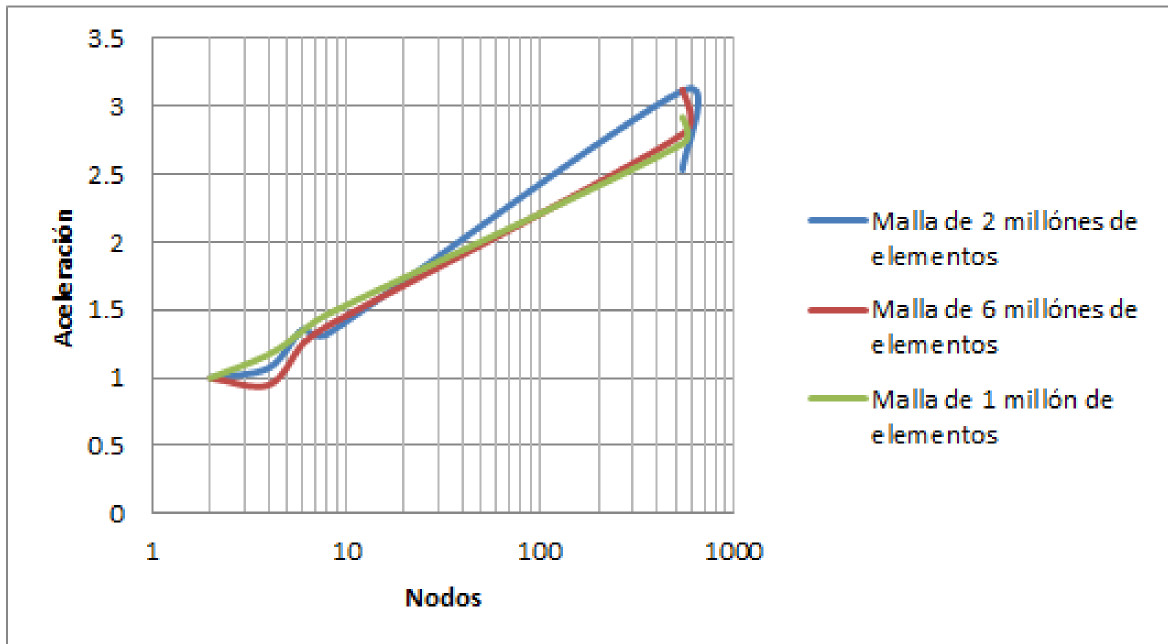

Figura 5.6: Aceleración contra el número de nodos

Se consideró una eficiencia del paralelismo unitaria para el caso de "2 cpu", ésto se realizó en mimetización de la teoría presentada en [27] y se puede observar en la Figura 5.7 donde se comienza la comparación desde el valor unitario, y que matemáticamente se puede representar:

$$\eta = \frac{t_n}{t_b * N} \tag{5.2}$$

Donde η es la eficiencia unitaria comparada contra el caso base, t_n es el tiempo de ejecución de *OpenFOAM* tomó para realizar 100 iteraciones, t_b es el tiempo de ejecución base o bien para 2 nodos registrados en *OpenFOAM*, para cada una de las 3 diferentes mallas en 100 iteraciones y N es el número de nodos. Cabe mencionar que el número de nodos para los casos de "gpu" y "gpu+multi" es de 240 nodos (*CUDA cores*).

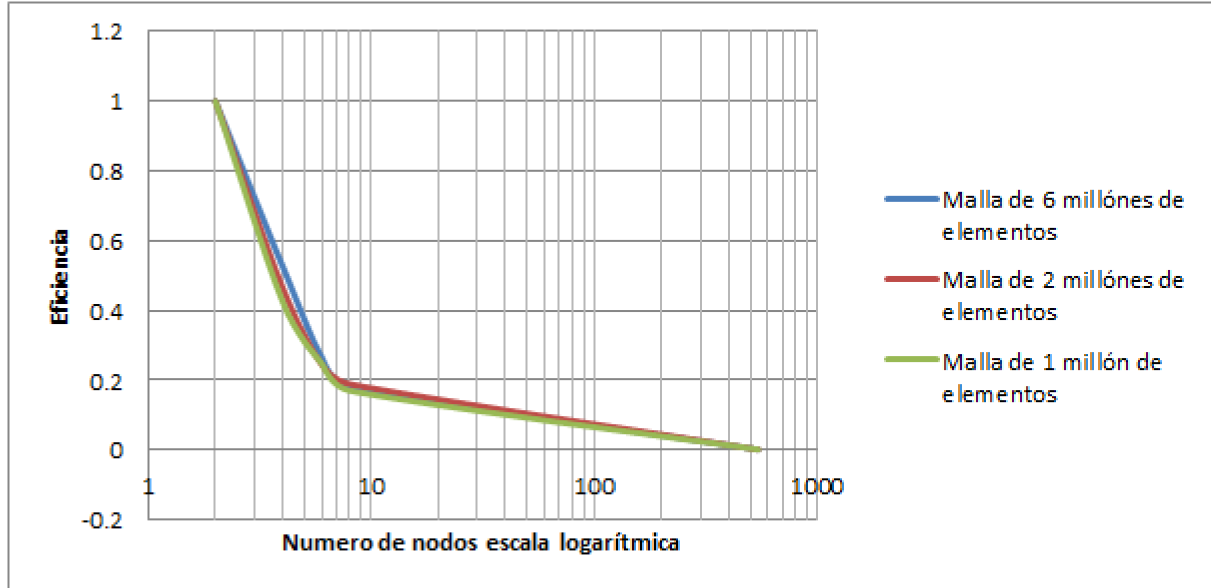

Figura 5.7: Eficiencia unitaria del paralelismo contra el número de nodos (caso base 2 CPU)

Las barras de porcentaje, Figura 5.8, presentan los valores porcentuales de ahorro de tiempo computacional. Se tomaron en cuenta en comparación a un caso base que es el que se realizó con más rapidez y que por tanto tiene el valor de cero. Por ejemplo, en la malla de 2 millones de elementos se ahorra 211 % del tiempo computacional haciendo el cálculo con el *GPU+single CPU* que sólo hacerlo con 2 nodos (2 *CPU*). Sin embargo el término es ambigüo, otro significado que se le podría dar gráficamente debido a la Fig. 5.8, siendo también el tiempo porcentual que tomaría aún más en realizar la misma tarea. Por ejemplo, en la malla de 1 millón de elementos para el caso de "2 cpu" se tomaría 192 % más de tiempo computacional del que realizó el caso "gpu+multi".

Para evaluar este ahorro se tiene:

$$\%ahorro = \frac{t_n}{t_f} * 100 \tag{5.3}$$

Donde $\%ahorro$ es el porcentaje de ahorro de tiempo computacional en contra del caso computacionalmente más rápido , $t_n$es el tiempo de ejecución de *OpenFOAM* que tomo para realizar 100 iteraciones y t_f es el tiempo de ejecución más rápido que *OpenFOAM* registra en los 6 casos enlistados en la tabla 5.3 también en 100 iteraciones.

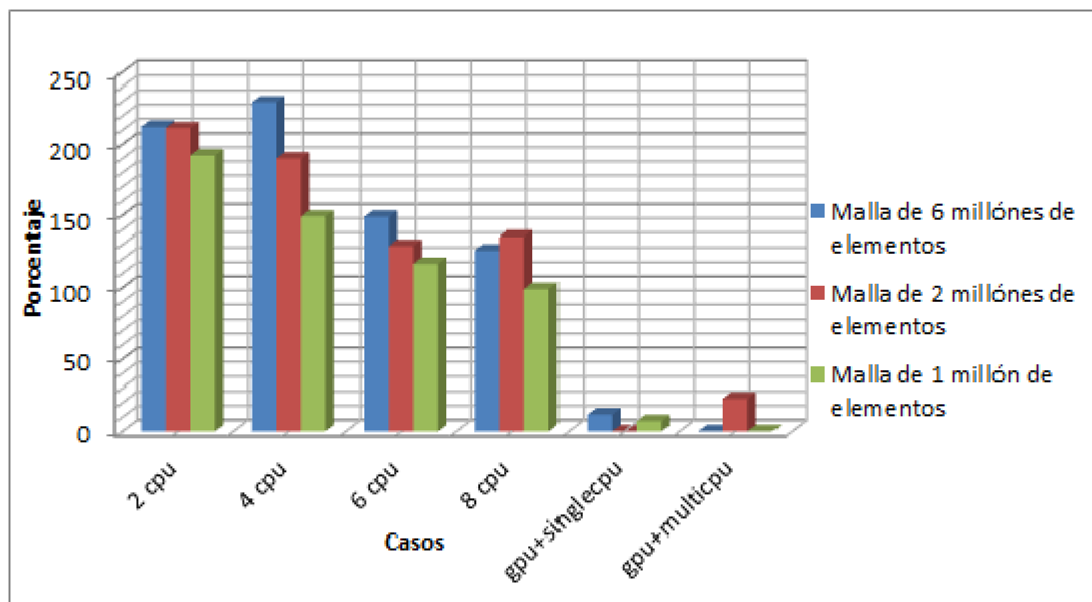

Figura 5.8: Porcentaje de ahorro de tiempo computacional en base al cálculo más rápido

5.4. Benchmark computacional en Fluent

Se realizò como trabajo de referencia un *benchmark* para medir el rendimiento computacional en uno de los simuladores de fluidos mas populares en la actualidad: *FLUENT* de *ANSYS*. Estos resultados no se incluyen en el *benchmark* objetivo (comparar rendimiento entre *GPU* y *CPU* en *OpenFOAM*) que se presentó en las secciones anteriores. Esto es debido a que sus resultados tendrian una diferencia en el rango de tiempo computacional por las diferencias en sus códigos, tales como son:

- Los esquemas de discretización y numéricos difieren en algunas partes del código. A pesar de que el código en *FLUENT* no es accesible, la teoría mencionada en el manual [40] es diferente a la de *OpenFOAM*, que se puede observar en su manual [38]. Un ejemplo es el esquema de discretización *Upwind* donde Fluent lo divide en dos (primer y segundo orden de discretización), mientras *OpenFOAM* solo usa uno.

- A pesar de que *OpenFOAM* se adapta por sus solucionadores y su entorno de programación para las necesidades individuales de los usuarios, es un *software* en desarrollo. Mientras que *FLUENT* es un *software* como lo señala [60] de *point and click*, con programadores listos para corregir los *bugs* que se presenten en el programa.

- Los criterios de convergencia no son los mismos entre los dos *software*. Esto quiere decir

que si *OpenFOAM* es configurado de manera correcta y monitoreados sus esquemas numéricos, se pueden hacer menos iteraciones para llegar al mismo grado de estabilidad de convergencia que realizaría *FLUENT* en una numerosa cantidad de iteraciones o viceversa.

Se presentan a continuación los resultados computacionales para 100 iteraciones con las mismas condiciones de frontera en donde las Figuras 5.9 y 5.10 muestran los tiempos computacionales usando un numero diferentes de nodos. Las Figuras 5.11 y 5.12 presentan la aceleracion y la eficiencia del paralelismo respectivamente.

Una conclusión importante de este pequeño *benchmark* adicional es que para obtener una comparación aproximadamente justa en parámetros computacionales, sería necesario adaptar los mismos criterios de convergencia de *FLUENT* a *OpenFOAM* o viceversa o bien realizar un estudio de convergencia con la estabilidad de los resultados y no con el número de iteraciones.

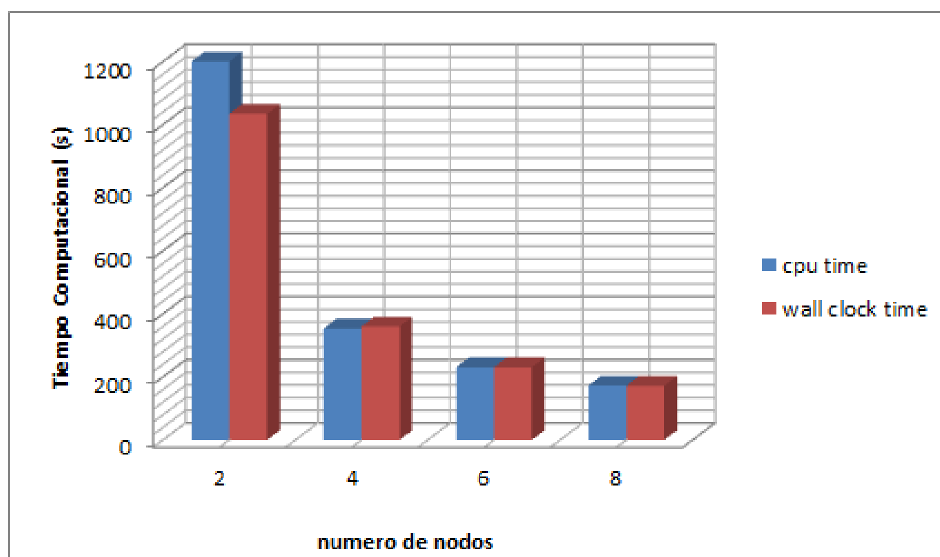

Figura 5.9: Gráfica de tiempos de cálculo para la malla de 1 millón de elementos en FLUENT

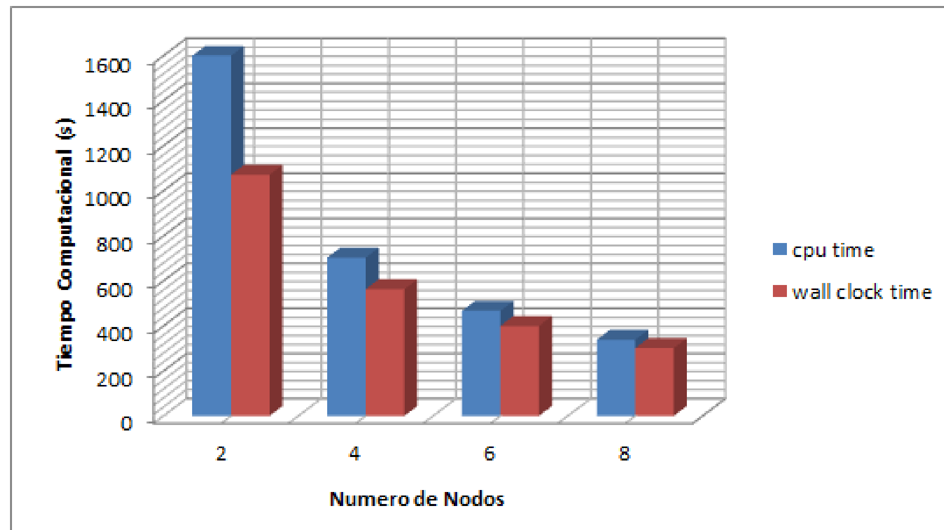

Figura 5.10: Gráfica de tiempos de cálculo para la malla de 2 millones de elementos en FLUENT

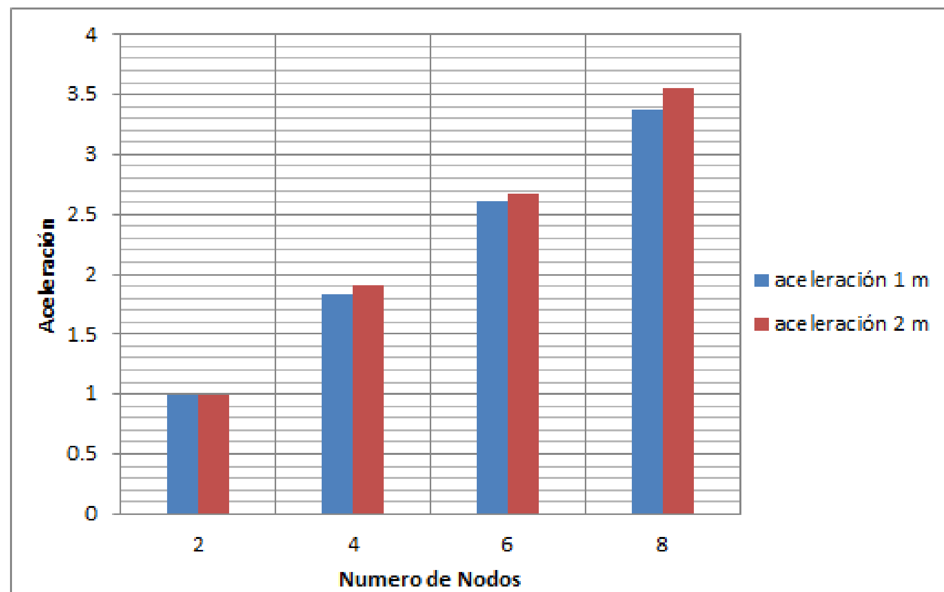

Figura 5.11: Aceleración contra el número de nodos en FLUENT

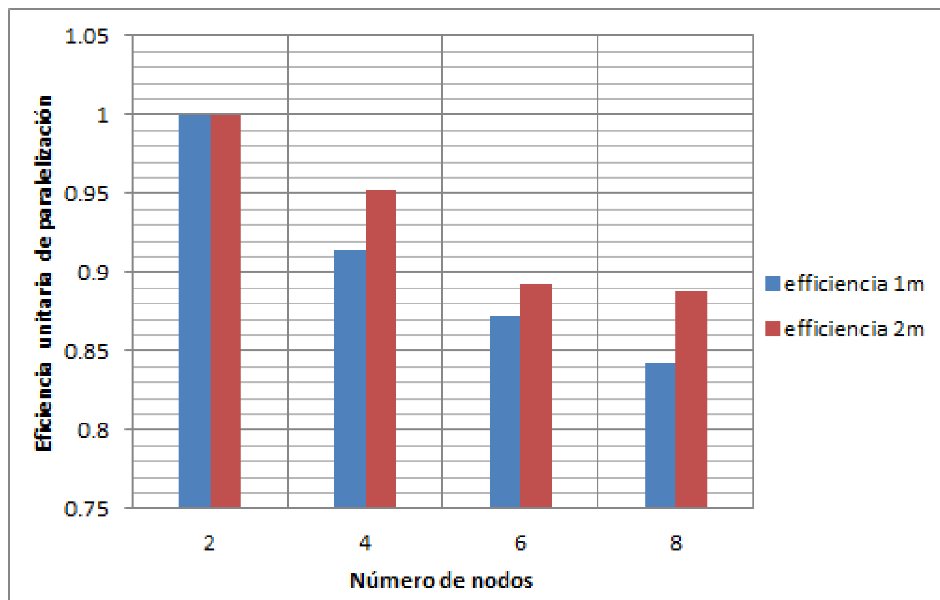

Figura 5.12: Eficiencia unitaria del paralelismo contra el número de nodos en FLUENT

Conclusiones de la validación *CFD* contra la experimental

Como se puede observar en las Fig. 4.5, el comportamiento del factor de recuperación de presión a lo largo de la pared superior e inferior del tubo de aspiración va acorde a los datos experimentales. Estas líneas tienden a aproximarse mucho al comportamiento experimental final del dispositivo.

Los resultados computacionales de la velocidad axial y de las líneas de corriente en las diferentes secciones del tubo, fueron comparados con el trabajo de Cervantes, [8, 26], (resultados presentados para el proyecto T-99). En la Figura 4.7 se presenta el comportamiento fluido dinámico que debe seguir el cono de expansión (fenómeno torcha) y en la Figura 4.6 se presenta la comparación de las velocidades axiales y de las líneas de corriente a lo largo del tubo. A pesar de manejar las mismas escalas y velocidades en cada sección, no es posible capturar exactamente los mismos contornos de velocidad.

Se puede observar en las Figuras 4.8 y 4.9 la validación contra resultados experimentales. La comparación de las simulaciones cuantitativa y cualitativamente con los resultados experimentales y con otros trabajos ya validados permite asegurar que la interpretación tanto de las condiciones de frontera como del modelo numérico han sido establecidas correctamente en el *software OpenFOAM*.

Conclusiones del *BenchMarck* Computacional

Se puede observar que el tiempo de ejecución y el tiempo de pared o real varían en un promedio de 5 %. Esta diferencia es aceptable considerando que la diferencia significa un mayor tiempo de procesamiento en los cálculos (Figuras 5.3, 5.4 y 5.5). Una diferencia significantiva entre el caso de "gpu" contra "gpu+multicpu" en la figura 5.3 en la malla de 2 millones de elementos es notable. Posiblemente se debe a algún cambio en el Sistema Operativo o errores del operador y se puede observar cómo inclusive el tiempo computacional difiere en su comportamiento en las gráficas 5.3 y 5.5 en contra de las pruebas en las mallas de 1 millón y 6 millones.

La hipótesis planteada en este trabajo de investigación es válida ya que el uso de procesadores gráficos como procesadores de los cálculos matriciales acelera los cálculos en un *software* de código abierto. Los resultados se resumen a continuación con referencia al máximo número de procesadores disponibles y al caso base de dos procesadores (para una descripción de los casos véase tabla 5.3) .

En la malla de 1 millón de elementos se tiene la siguiente aceleración:

- En relación al caso base (uso de 2 procesadores) "2cpu", 2.72 veces más rápido usando el caso de "gpu" y 2.9 veces usando el caso de "gpu+multi".
- En relación al uso de 8 procesadores (caso "8 cpu") se tiene una aceleración de 1.85 usando el caso de "gpu" y 1.98 veces usando el caso de "gpu+multi".

En la malla de 2 millones de elementos se observa también un aumento de la aceleración usando el GPU:

- En relación al caso base 2cpu, 2.53 veces usando el caso de "gpu" y 2.36 usando el caso de "gpu +multi" .

- Comparando contra el caso de "8cpu" se aceleró 1.91 veces usando el caso de "gpu" y 2.35 veces más rápido usando el caso de "gpu +multi".

En la malla de 6 millones de elementos se observa:

- En relación al caso base "2 cpu" se acelera 3.12 veces usando el caso de "gpu" y 3.29 con el caso de "gpu +multi".
- Comparando contra el uso de "8 cpu", 2.02 veces más rápido usando el caso de "gpu" y 2.25 veces más rápido usando el caso de "gpu +multi".

Los porcentajes de ahorro muestran una eficiencia de tiempo de ahorro computacional del GPU de hasta 200 %, Fig. 5.8, en contra de sólo usar el CPU. Pero en contraste la eficiencia del paralelismo es reducida hasta 0.01, Fig. 5.12. Esto es debido a que los núcleos de CPU y GPU no tienen las mismas capacidades computacionales y también es debido a que la comunicación entre el *host* (*CPU*) y el dispositivo (*GPU*) o bien proceso *CUDA*, Fig. 2.3, no es eficiente y gasta tiempo computacional valioso.

Trabajo a Futuro

Este trabajo de investigación permitió adaptar un solo *GPGPU* para acelerar los cálculos en el benchmarck del tubo de aspiración de la Turbina-99. De los resultados obtenidos en este trabajo se plantean a continuación tres líneas de investigación que podrían lograr una mejor implementación del cálculo de alto rendimiento en flujos industriales:

- ANÁLISIS DE TURBULENCIA: El tiempo computacional siempre es crucial y en el uso de modelos de turbulencia más complejos y con menos idealizaciones se requiere una malla más fina lo que limitaría la aceleración de sus cálculos. Es interesante que con el poder de dispositivos *GPGPU* se puedan realizar modelizaciones de la turbulencia más complejas del tubo de aspiración T-99 o bien prescindir de estos modelos para realizar un estudio fluido dinámico por medio de métodos directos (*DNS*) y estudiar detalladamente sus variables y constantes que se usan en la configuración fluido dinámica computacional.

- HPC A GRAN ESCALA: La evolución de los ordenadores es constante y el uso de herramientas más sofisticadas para el *HPC* es una opción interesante; como puede ser la combinación de un clúster de ordenadores haciendo sus cálculos no sólo en los *CPU*s si no incluyendo el co-procesamiento en sistemas de *multiGPU*s. También cabe destacar la posibilidad de usar *GPU*s de altas prestaciones como es la tecnología *Kepler* que sería una interesante opción no sólo por tener aproximadamente 10 veces más la cantidad de *CUDA cores* (*Tesla K20*) que la *TESLA C1060*, Fig. 5.2, sino por todas sus altas especificaciones y sus altas prestaciones que este dispositivo ofrece. Estos estudios *en CFD* usando este tipo de nuevas tecnologías *HPC* se pueden aplicar no sólo al dispositivo del proyecto T-99 sino a otros fenómenos interesantes.

- MALLA Y ESQUEMAS NUMÉRICOS Y DE DISCRETIZACIÓN PARA LA T-99: Como trabajo posterior se puede realizar un estudio de malla y las diferentes formas de discretización del dominio del problema T-99. La correcta discretización del dominio

es crítica y un análisis a fondo de cómo este parámetro está relacionado al tiempo de cálculo es posible. Se observa en la sub-sección del capítulo 5: benchmark computacional en FLUENT, qué tan importantes son los esquemas numéricos y la estabilidad de la convergencia para reducir el tiempo computacional. Es por ello que como perspectiva a futuro, un estudio de cómo estos parámetros afectan el tiempo computacional es una interesante opción.

Conclusión Final

El flujo en el tubo de aspiración (T-99) es un ejemplo de los problemas que son un reto en aplicaciones industriales donde se realizan estudios con *CFD* y la caja de herramientas *OpenFOAM* ha demostrado ser una buena plataforma para obtener cálculos de alta calidad.

Uno de los principales beneficios de utilizar *OpenFOAM* en lugar de cualquier *software* comercial es que el código fuente completo está abierto y disponible. Esto hace posible el desarrollo de métodos que están fuera del alcance de los programas comerciales de *CFD*.

En consecuencia, la principal contribución de este trabajo ha sido que al poder implementar la aplicación de un sólo *GPU* en *OpenFOAM* ha demostrado acelerar el costo computacional en una cantidad importante. El siguiente paso es saber si la implementación de varios *GPU*'s sobre un mismo *CPU* o si un solo *GPU* es la mejor opción para acelerar cálculos en este tipo de flujos industriales tan complicados para simular. Finalmente la ciencia de los cálculos debe ser estudiada a detalle tanto numéricamente como computacionalmente.

Bibliografía

[1] Guan H. Y. and Jiyuan T. Computational Fluid Dynamics: A Practical Approach. s.l. Butterworth-Heinemann. 2007.

[2] Agüera. J. S. Mecánica de Fluidos Incompresibles y Turbomáquinas Hidráulicas, 5th. edition. 2002.

[3] Universidad de Castilla La Mancha, Ingeniería Técnica Industrial.[En línea]. http://usuarios.multimania.es/jrcuenca/Spanish/esp_INICIO.htm. [Fecha de acceso 25 de Octubre del 2012].

[4] Andersson U. Experimental study of sharp-heel Kaplan draft tube (doctoral thesis). Luleå : Luleå University of Technology. 2009.

[5] Jonsson P. Measurements in the U9 Kaplan turbine model (doctoral thesis). Luleå : Luleå University of Technology. 2011.

[6] Engström T.F.Turbine-99 workshop on draft tube flow. 2000.

[7] Engström T.F.; Gustavsson L.H and Karlsson R.I. The second ERCOFTAC Workshop on Draft Tube Flow. Sweden. 2001.

[8] Cervantes M.J.; Engström T.F.; Gustavsson L.H. Proceedings of the third IAHR/ERCOFTAC workshop on draft tube flows Turbine-99 III. 2005.

[9] Nilsson H. and Page M. OpenFOAM Simulation of the flow in the Hölleforsen draft tube model. In: Proceedings of Turbine-99 III. 2005

[10] Petit O.; Nilsson H.; Thi Vu; Manole O. and Leonsson S. Chalmers and Andritz V.A. Tech Hydro Ltd. The flow in the U9 kaplan turbine.Preliminary and planned simulations using CFX and Openfoam. In: Proceedings of the 24rd IAHR Symposium. 2008.

[11] Nilsson H. Evaluation of OpenFOAM for CFD of turbulent flow in water turbines. In: Proceedings of the 23rd IAHR Symposium in Yokohama. 2006.

[12] Galvan S.R; Reggio M. and Guibalt F. Assessment study of k-e turbulence models and near- wall modeling for steady state swirling flow analysis in a draft tube using Fluent. In Enginnering Application of Computational Fluid Mechanics.2011.

[13] Andersson U. Turbine 99 -Experiments on draft tube flow (test case T). In: Proceedings of Turbine 99 -Workshop on Draft Tube Flow, ISSN: 1402 -1536. 2000.

[14] Nilsson H. Some Experiences on the Accuracy and Parallel Performance of OpenFOAM for CFD in Water Turbines. Springer-Verlag Berlin, Heidelberg, ISBN: 3-540-75754-6 978-3-540-75754-2. 2007.

[15] Corrigan A.; Camelli F.; Lohnerz R. and Wallin J. Running Unstructured Grid Based CFD Solvers on Modern Graphics Hardware. In 19th AIAA Computational Fluid Dynamics.USA.2009.

[16] Stantchev G; Juba D; Dorland W; and Varshney A. High-performance Computation and Visualization of Plasma Turbulence on Graphics Processors. In Computing in Science and Engineering archive, Volume 11 Issue 2. 2009.

[17] Cohen J. M. and Molemaker M. J.SNVIDIA Corporation and IGPP UCLA. A Fast Double Precision CFD Code using CUDA. In Proceedings of Parallel Computational Fluid Dynamics, CA 95050. USA. 2009.

[18] Hai Tuan Nguyen. Compressible NavierStokes Solver with openSource for graphical processing Units (Master Thesis). San Jose State University, USA. 2012.

[19] Henry W. OpenCFD Ltd.[En línea] 2004.www.k enfoam.com. [Fecha de Acceso 8 de Febrero 2012].

[20] Mulu B and Cervantes M. J.Experimental investigation of a Kaplan model with LDA. In International Association of Hydraulic Engineering & Research (IAHR), ISBN: 978-94-90365-01-1. 2009.

[21] Björn J. Evaluation of CFD Model for Kaplan DraftTube (doctoral thesis). Luleå University of Technology. 2012.

[22] Page M. (Hydro-Québec); Beaudoin. M. (Hydro-Québec) and Nilsson H. (Chalmers).[En línea]. http://openfoamwiki.net/index.php/Sig_Turbomachinery.[Fecha de acceso 25 de Octubre del 2012].

[23] Avellan F. Flow investigation in a Francis draft tube. The FLINDT Project. In: Proceedings of 20th IAHR symposium on hydraulic machinery and system. Charlotte USA. 2000.

[24] Hellström J.G.I.; Marjavaara B.D. and Lundström T.S. Parallel CFD simulations of an original and redesigned hydraulic turbine draft tube. Advances in Engineering Software, 338–344. 2007 .

[25] Cervantes M.J.. Effects of boundary conditions and unsteadiness on draft tube flow (doctoral thesis). Luleå : Luleå University of Technology. 2003.

[26] Cervantes M.J. and Engström T.F. Eddy viscosity turbulence models and steady draft tube simulations. In Proceedings of the third IAHR/ERCOFTAC workshop on draft tube flow. Sweden, Porjus. 2005.

[27] László N.; Péter T.; Mátě M. L. and Ákos C. Fluent benchmark on an amd athlon64 single and dual core white box hpc cluster. In Fluid and Heat Engineering, pages:43-48, Proceedings of MicroCAD'2008, Miskolc, Hungary. March 2008.

[28] Daugherty R.L.; Franzini J.B. and Finnemore, E.J.. Fluid Mechanics with engineering applications, McGraw-Hill Book Company, Singapore. 1989.

[29] Stano Y.; Kawaguchi N. and Tetsuzou, T. Swirl flow in conical diffusers, Bulletin of the JSME, Vol. 21., No. 151., January, pp. 112-119. 1978.

[30] Staubli T. and Deniz S. Analysing draft tube kinetics. Int. Water Power & Dam Construction, December, pp. 38-42. 1994.

[31] Bardina J.E.; Huang P.G. and Coakley T.J. Turbulence Modeling Validation, Testing, and Development. In NASA Technical Memorandum 110446.1997.

[32] Jones W. P.; and Launder B. E. The Prediction of Laminarization with a Two-Equation Model of Turbulence.In International Journal of Heat and Mass Transfer, vol. 15, 1972, pp. 301-314. 1972.

[33] Launder B. E. and Sharma, B. I. Application of the Energy Dissipation Model of Turbulence to the Calculation of Flow Near a Spinning Disc. Letters in Heat and Mass Transfer, vol. 1, no. 2, pp. 131-138. 1974.

[34] Wilcox D. C. Turbulence Modeling for CFD. Second edition. Anaheim: DCW Industries, 1998. pp. 174. 1998.

[35] Vaidehi A. Simple solver for driven cavity flow problem. Department of Mechanical Engineering Purdue University.Indiana USA. 2006.

[36] Malecha Z.; Mirosław L.; Tomczak T.; Koza Z.; Matyka M.St Tarnawski W. and Szczerba D. GPU-based simulation of 3D blood flow in abdominal aorta using OpenFOAM. In Arch. Mech., 63, 2, pp. 137–161, Warszawa. 2011

[37] Fernández O. Jesús M. Técnicas numéricas en ingeniería de fluido: introducción a la dinámica de fluidos computacional (CFD) por el método de volúmenes finitos. Editorial Reverté. Barcelona 2012.

[38] OpenFOAM. The Open source CFD toolbox programmer's guide. Version 2.0. 2012.

[39] NVIDIA CUDA Zone. [en línea]. http://www.nvidia.com/object/cuda_home_new.html. 2012.

[40] PathScale Corporation. Fluent 6.3 user's guide [en línea] http://aerojet.engr.ucdavis.edu/fluenthelp/pdf/ug/pdf-pdf-download.htm. 2012.

[41] Issa R. . Solution of implicitly discretized fluid flow equations by operator-splitting, J. Comput. Phys. 62, 40–65. 1986.

[42] Chung T. Computational fluid dynamics, Cambridge University Press. 2002.

[43] Standard J. and Perić M. Computational methods for fluid dynamics, Springer Berlin. 1999.

[44] Versteeg H. and W. Malalsekera. An introduction to computational fluid dynamics, Longman Scientific & Technical. 1995.

[45] NVIDIA CUDA Programming Guide Version 3.0, NVIDIA, 2010. [en línea] http://deve- loper.download.nvidia.com /compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf.

[46] ATI Stream Technology, http://www.amd.com/us/products/technologies/stream-technology/Pages/stream-technology.aspx. 2009.

[47] Tölke J. and Krafczyk M. TeraFLOP computing on a desktop PC with GPUs for 3D CFD, International Journal of Computational Fluid Dynamics, 22, 7, 443–456. 2008

[48] Elsen E.; LeGresley P. and Darve E. Large calculation of the flow over a hypersonic vehicle using a GPU, J. Comput. Phys., 227, 24, 10 148–10 161. 2008.

[49] Thibault J.C. and Senocak I. CUDA Implementation of a Navier–Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows. In 47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition, American Institute of Aeronautics and Astronautics, Inc. 2009.

[50] Senocak I.; Thibault J. and M. Caylor. Rapid Response Urban CFD Simulations Using a GPU Computing Paradigm on Desktop Supercomputers. In Eighth Symposium on the Urban Environment.field. 2010.

[51] Jacobsen D.A.; J.C. Thibault and I. Senocak. An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. In 48th AIAA Aerospace Sciences Meeting, American Institute of Aeronautics and Astronautics, Inc. 2010.

[52] Göddeke D. GPU acceleration of an unmodified parallel finite element Navier- Stokes solver. High Performance Computing & Simulation. 2009.

[53] Fletcher C. . Computational Techniques for Fluid Dynamics 2, Springer-Verlag. 1991.

[54] Hoffmann K. and S. Chiang. Computational Fluid dynamics, Vol. II, Engineering Education System. 2000.

[55] Saad. Y. Iterative Methods for Sparse Linear Systems, SIAM. 2003.

[56] Barrett R. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, Philadelphia, PA: SIAM. 1994.

[57] Bell N. and Garland M. Implementing sparse matrix-vector multiplication on throughput oriented processors. In SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 1–11, New York, NY, USA, ACM. 2009.

[58] Davis T.A. and Y. Hu, The University of Florida Sparse Matrix Collection, submitted to ACM Transactions on Mathematical Software.

[59] Christ A. The virtual family-development of surface-based anatomical models of two adults and two children for dosimetric simulations, Physics in Medicine and Biology, 55, 2, N23. 2010.

[60] Andersen C. and Nielsen N. E. L. Numerical investigation of a BFR using OpenFOAM. Report for AAU - Institute of Energy Technology. 2008.

Apéndice A

I. Geometría T-99

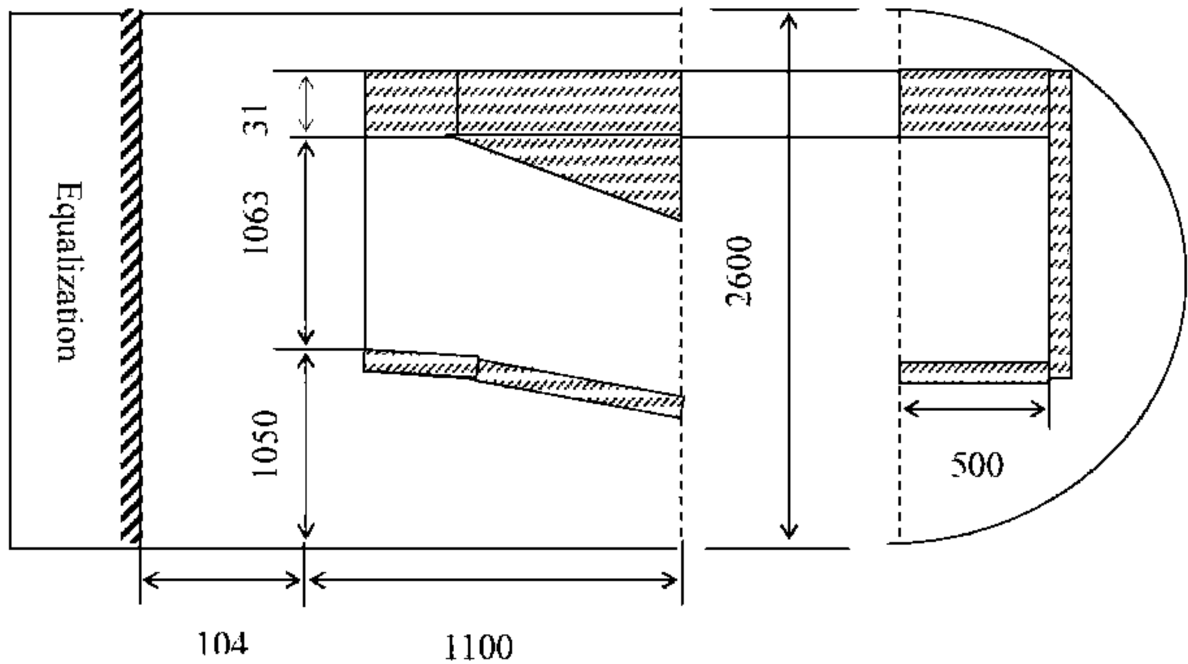

Figura A1: Dimensiones del depósito de agua. El tubo de aspiración termina en un depósito cilíndrico grande y que tiene un diámetro de 2,600 mm, y en el centro de la salida del tubo de aspiración está situado a 282 mm por encima del centro del cilindro [7, 8].

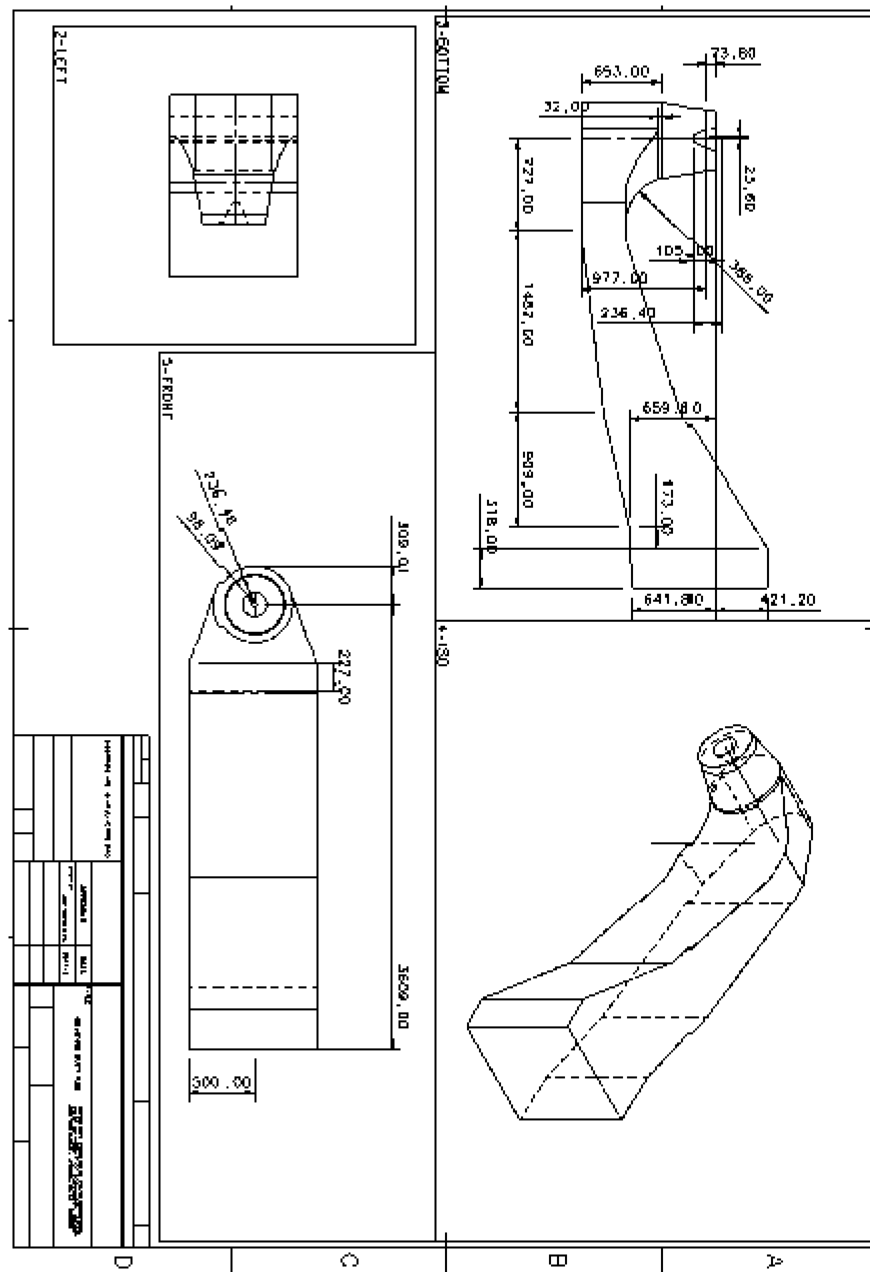

Figura A2: Dibujos del tubo de aspiración T-99 [7, 8].

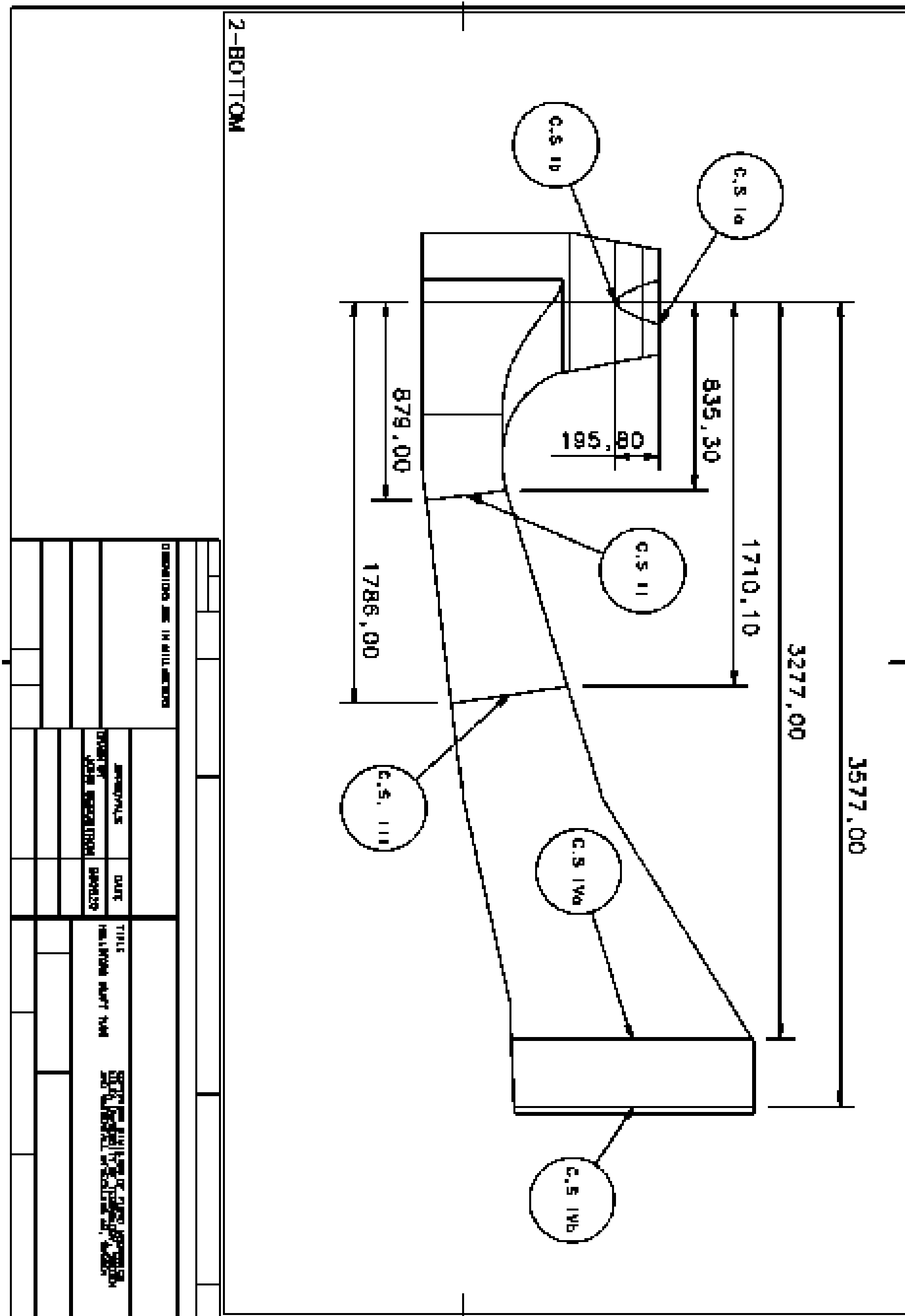

Figura A3: Localización de las secciones de medición [7, 8].

II. Malla OpenFOAM

La descripción de las mallas se encuentra en formato texto y parámetros enlistados como calidad, número de elementos, etc, se encuentran descritos por la aplicación checkMesh

Create time
Create polyMesh for time = 0
Time = 0
Mesh stats
all points: 1002360
live points: 1002360
all faces: 2964987
live faces: 2964987
internal faces: 2923557
cells: 981424
boundary patches: 4
point zones: 0
face zones: 7
cell zones: 1
Overall number of cells of each type:
hexahedra: 981424
prisms: 0 wedges: 0 pyramids: 0 tet wedges: 0 tetrahedra: 0 polyhedra: 0
Checking topology...
Boundary definition OK.
Point usage OK.
Upper triangular ordering OK.
Face vertices OK.
Number of regions: 1 (OK).
Checking geometry...
This is a 3-D mesh
Overall domain bounding box (-0.30901498 -0.50000006 -1.0508001) (3.609 0.49999994 0.42120001)
Mesh (non-empty, non-wedge) directions (1 1 1) Mesh (non-empty) directions (1 1 1) Mesh (non-empty, non-wedge) dimensions 3
Boundary openness (-2.940783e-16 3.2432548e-16 1.5860946e-16) Threshold = 1e-06 OK.
Max cell openness = 5.4411848e-15 OK. Max aspect ratio = 85.274947 OK.
Minumum face area = 1.Standard-07. Maximum face area = 0.0021072582.
Face area magnitudes OK. Min volume = 1.8830897e-10. Max volume = 5.1575875e-05. Total volume = 2.4360718. Cell volumes OK.
Mesh non-orthogonality Max: 77.953898 average: 19.084416 Threshold = 70
**Number of severely non-orthogonal faces: 123. Non-orthogonality check OK.*
Writing 123 non-orthogonal faces to set nonOrthoFaces Face pyramids OK.
Max skewness = 2.8379596 OK.
Mesh OK.
End

Figura A4: Salida texto de la aplicación checkMesh para la malla de 1 millón de elementos

Create time
Create polyMesh for time = 0
Time = 0
Mesh stats
all points: 2263550
live points: 2263550
all faces: 6709545
live faces: 6709545
internal faces: 6629247
cells: 2223132
boundary patches: 6
point zones: 0
face zones: 8
cell zones: 1
Overall number of cells of each type:
hexahedra: 2223132
prisms: 0 wedges: 0 pyramids: 0 tet wedges: 0 tetrahedra: 0 polyhedra: 0
Checking topology...
Boundary definition OK.
Point usage OK.
Upper triangular ordering OK.
Face vertices OK.
Number of regions: 1 (OK).
Checking geometry...
This is a 3-D mesh
Overall domain bounding box (-0.309015 -0.5 -1.0508) (3.609 0.5 0.4212)
Mesh (non-empty, non-wedge) directions (1 1 1) Mesh (non-empty) directions (1 1 1) Mesh (non-empty, non-wedge) dimensions 3
Boundary openness (-6.18828e-17 -2.70429e-15 -3.61932e-15) Threshold = 1e-06 OK.
****High aspect ratio cells found, Max aspect ratio: 4274.52, number of cells 31694 Threshold = 1000*
Writing 31694 cells with high aspect ratio to set highAspectRatioCells
Minumum face area = 1.12265e-09. Maximum face area = 0.00210567.
Face area magnitudes OK. Min volume = 5.80778e-13. Max volume = 2.53019e-05. Total volume = 2.43631. Cell volumes OK.
Mesh non-orthogonality Max: 80.1037 average: 19.Plai Threshold = 70
**Number of severely non-orthogonal faces: 946. Non-orthogonality check OK.*
Writing 946 non-orthogonal faces to set nonOrthoFaces Face pyramids OK.
Max skewness = 2.42713 OK.
Failed 1 mesh checks.
End

Figura A5: Salida texto de la aplicación checkMesh para la malla de 2 millones de elementos

Create time
Create polyMesh for time = 0
Time = 0
Mesh stats
all points: 5691841
live points: 5691841
all faces: 16892739
live faces: 16892739
internal faces: 16711077
cells: 5600636
boundary patches: 6
point zones: 0
face zones: 9
cell zones: 2
Overall number of cells of each type:
hexahedra: 5600636
prisms: 0 wedges: 0 pyramids: 0 tet wedges: 0 tetrahedra: 0 polyhedra: 0
Checking topology...
Boundary definition OK.
Point usage OK.
Upper triangular ordering OK.
Face vertices OK.
Number of regions: 1 (OK).
Checking geometry...
This is a 3-D mesh Overall
domain bounding box (-0.309015 -0.5 -1.0508) (3.609 0.5 0.4212)
Mesh (non-empty, non-wedge) directions (1 1 1) Mesh (non-empty) directions (1 1 1) Mesh (non-empty, non-wedge) dimensions 3
Boundary openness (4.88888e-16 1.70037e-15 -1.39431e-15) Threshold = 1e-06 OK.
****High aspect ratio cells found, Max aspect ratio: 4274.52, number of cells 31787 Threshold = 1000*
Writing 31787 cells with high aspect ratio to set highAspectRatioCells
Minumum face area = 1.02801e-09. Maximum face area = 0.00210567.
Face area magnitudes OK. Min volume = 2.xte98e-13. Max volume = 1.26518e-05. Total volume = 2.43634. Cell volumes OK.
Mesh non-orthogonality Max: 80.1037 average: 17.1887 Threshold = 70
**Number of severely non-orthogonal faces: 1062. Non-orthogonality check OK.*
Writing 1062 non-orthogonal faces to set nonOrthoFaces Face pyramids OK.
Max skewness = 1.98085 OK.
Failed 1 mesh checks.
End

Figura A6: Salida texto de la aplicación checkMesh para la malla de 6 millones de elementos

Apéndice B

I. Configuración para los solucionadores

```
solvers
{
pcorr
{
solver cufflink_DiagPBiCGStab; //BiCGStab;
preconditioner diagonal;
storage 2;
gpusPerMachine 1;
minIter 0;
maxIter 1000;
tolerance 1e-12;
// relTol 0;
}
p
{
solver cufflink_CG; //BiCGStab;
preconditioner DIC;
tolerance 1e-12;
storage 2;
gpusPerMachine 1;
minIter 0;
maxIter 1000;
// relTol 0;
}
```

```
pFinal
{
solver cufflink_DiagPBiCGStab; //BiCGStab;
preconditioner diagonal;
storage 2;
gpusPerMachine 1;
minIter 0;
maxIter 1000;
tolerance 1e-12;
// relTol 0;
}
U
{
solver cufflink_DiagPBiCGStab; //BiCGStab;
preconditioner diagonal;
storage 2;
gpusPerMachine 1;
minIter 0;
maxIter 1000;
tolerance 1e-12;
// relTol 0;
}
ensilon
{
solver cufflink_DiagPBiCGStab; //BiCGStab;
preconditioner diagonal;
storage 2;
gpusPerMachine 1;
minIter 0;
maxIter 1000;
tolerance 1e-12;
// relTol 0;
}
k
{
```

```
    solver cufflink_DiagPBiCGStab; //BiCGStab;
    preconditioner diagonal;
    storage 2;
    gpusPerMachine 1;
    minIter 0;
    maxIter 1000;
    tolerance 1e-12;
    // relTol 0;
    }
    UFinal
    {
    solver cufflink_DiagPBiCGStab; //BiCGStab;
    preconditioner diagonal;
    storage 2;
    gpusPerMachine 1;
    minIter 0;
    maxIter 1000;
    tolerance 1e-12;
    // relTol 0;
    }
    }
SIMPLE
{
nNonOrthogonalCorrectors 0;
pRefCell 0;
pRefValue 0;
}
relaxationFactors
{
p 0.1;
U 0.5;
k 0.5;
epsilon 0.5;
}
```

Figura B1: Configuración del caso para los *solvers (fvSolution)*

II. Configuración para los esquemas numéricos

```
ddtSchemes
{
default steadyState; //Euler;
}
gradSchemes
{
default Gauss linear;
grad(p) Gauss linear;
}
divSchemes
{
default none;
div(phi,U) Gauss linearUpwind Gauss; //upwind;
div((nuEff*dev(grad(U).T()))) Gauss linear;
div(phi,epsilon) Gauss upwind;
div(phi,k) Gauss upwind;
}
laplacianSchemes
{
default none;
laplacian(nu,U) Gauss linear corrected;
laplacian(rAU,pcorr) Gauss linear corrected;
laplacian(rAU,p) Gauss linear corrected;
laplacian(nuEff,U) Gauss linear corrected;
laplacian((1|A(U)),p) Gauss linear corrected;
laplacian(DepsilonEff,epsilon) Gauss linear corrected;
laplacian(DkEff,k) Gauss linear corrected;
laplacian(1,p) Gauss linear corrected;
}
interpolationSchemes
{
default linear;
interpolate(HbyA) linear;
```

interpolate(1|A) linear;

}

snGradSchemes

{

default corrected;
}
fluxRequired
{
default no;
pcorr;
p;
}

Figura B2: Configuración del caso para los esquemas de discretización y numéricos (*fvSchemes*)

III. Configuración para el control de la base de datos

application icoTopoFoam;

startFrom startTime;

startTime 0;

stopAt endTime;

endTime 5000;

deltaT 1;

writeControl timeStep;

writeInterval 100;

cycleWrite 0;

writeFormat ascii;

writePrecision 6;

writeCompression compressed;

timeFormat general;

timePrecision 6;

runTimeModifiable yes;

adjustTimeStep yes;

maxDeltaT 1.0;

functions

```
(
trackDictionaryContent
{
type trackDictionary;
// Where to load it from (if not already in solver)
functionObjectLibs ("libsimpleFunctionObjects.so");
// Names of dictionaries to track.
dictionaryNameList
(
"system/controlDict"
"system/fvSchemes"
"system/fvSolution"
"constant/transportProperties"
"constant/RASProperties"
);
// Section separators (optional)
// If the string "_sectionIdToken_" explicitely appears in the
// specification of the following section separators, this token
// string will be replaced by the name of the dictionary beeing
// dumped to the console, plus the file modification date and time.
sectionStartSeparator "############
Start of: _sectionIdToken_ ############";
sectionEndSeparator "############
End of: _sectionIdToken_ ############";
//
echoControlDictDebugSwitches off;
echoControlDictInfoSwitches off;
echoControlDictTolerances off;
```

```
echoControlDictOptimisationSwitches off;
echoControlDictDimensionedConstants off;
}
);
libs ("libCufflink.so" "libOpenFoamTurbo.so");
```

Figura B3: Configuración del caso para el control de entrada y salida o también llamada I/O que establece parámetros esenciales para la creación de la base de datos de entrada como son las librerías.

IV. Configuración para las condiciones de frontera de épsilon

```
dimensions [ 0 2 -3 0 0 0 0 ];
internalField uniform 0.1;
boundaryField
{
paredes
{
type zeroGradient;
}
entrada
{
fieldName "Epsilon";
fileFormat "turboCSV";
fileName "case1_Tn.W3.Umoy3p59.csv";
interpolateCoord "R";
type profile1DfixedValue;
}
salida
{
type zeroGradient;
}
```

```
cono
{
type zeroGradient;
}
}
```

Figura B4: Configuración del caso para la disipación energética o épsilon (*epsilon*)

V. Configuración para las condiciones de frontera de kappa

```
dimensions [ 0 2 -2 0 0 0 0 ];
internalField uniform 0.1;
boundaryField
{
paredes
{
type zeroGradient;
}
entrada
{
fieldName "K";
fileFormat "turboCSV";
fileName "case1_Tn.W3.Umoy3p59.csv";
interpolateCoord "R";
type profile1DfixedValue;
}
salida
{
```

```
type zeroGradient;
}
cono
{
type zeroGradient;
}
}
```

Figura B5: Configuración del caso para la energía cinética (k)

VI. Configuración para las condiciones de frontera de presión

```
dimensions [ 0 2 -2 0 0 0 0 ];
internalField uniform 0;
boundaryField
{
paredes
{
type zeroGradient;
}
entrada
{
type zeroGradient;
}
salida
{
type fixedValue;
value uniform 0;
}
cono
{
type zeroGradient;
}
}
```

Figura B6: Configuración del caso para el campo vectorial de presiones (P)

VII. Configuración para las condiciones de frontera de velocidad

```
dimensions [ 0 1 -1 0 0 0 0 ];
internalField uniform (0 0 -1);
boundaryField
{
paredes
{
type fixedValue;
value uniform (0 0 0);
}
entrada
{
fieldName "Velocity";
fileFormat "turboCSV";
fileName "case1_Tn.W3.Umoy3p59.csv";
interpolateCoord "R";
type profile1DfixedValue;
}
salida
{
type zeroGradient;
}
cono
{
type fixedValue;
value uniform (0 0 0);
}
}
```

Figura B7: Configuración del caso para el campo vectorial de velocidad (U)

VIII. Codigo para ejecutar *Solver* en *CUDA*

*Solucionador para los cálculos de matrices asimétricas

```
#include "cufflink_DiagPBiCGStab.H"
//for COO and vector on host
#include <cusp/coo_matrix.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/copy.h>
#include "../CFL_Headers/cuspTypeDefs.H"
//end headers for COO and vectors
#include <iomanip>
#include <iostream>
#include <string>
#include <stdio.h>
//using namespace std;
#include "../CFL_Headers/OFSolverPerformance.H"
#include "../CFL_Headers/cusp_equation_system.H"
extern "C" void CFL_DiagPBiCGStab(cusp_equation_system *CES, OFSolverPerformance *OFSP);//extern function in CUDA that solves the equation system
// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //
namespace Foam
{
defineTypeNameAndDebug(cufflink_DiagPBiCGStab, 0);
lduSolver::addasymMatrixConstructorToTable<cufflink_DiagPBiCGStab>
addcufflink_DiagPBiCGStabAsymMatrixConstructorToTable_;
}
// * * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * * * * //
Foam::cufflink_DiagPBiCGStab::cufflink_DiagPBiCGStab
(
const word& fieldName,
const lduMatrix& matrix,
const FieldField<Field, scalar>& coupleBouCoeffs,
const FieldField<Field, scalar>& coupleIntCoeffs,
const lduInterfaceFieldPtrsList& interfaces,
const dictionary& dict
)
```

```
:
lduSolver
(
fieldName,
matrix,
coupleBouCoeffs,
coupleIntCoeffs,
interfaces,
dict
)
{}
// * * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * * * //
Foam::lduSolverPerformance Foam::cufflink_DiagPBiCGStab::solve
(
scalarField& x,
const scalarField& b,
const direction cmpt
) const
{
OFSolverPerformance OFSP;//container for solver performance
//get the sparse matrix storage method from fvSolution
#include "../CFL_Headers/getGPUStorage.H"
// — Setup class containing solver performance data
lduSolverPerformance solverPerf
(
"cufflink_DiagPBiCGStab",
fieldName()
);
//insert my solver code here
register label nCells = x.size();
register label NFaces = matrix().lower().size();
OFSP.nCells = nCells; OFSP.nFaces = NFaces; OFSP.debugCusp = false;
//turn on the debug switch to print some important information
if (lduMatrix::debug >= 2)
{OFSP.debugCusp = true;
```

```
}
cusp_equation_system CES;
//declare the COO, X and b and copy the OpenFOAM values here
CES.A = cusp::coo_matrix<IndexType, ValueType, hostMemorySpace>
(nCells,nCells,nCells+2*NFaces);
CES.X = cusp::array1d< ValueType, hostMemorySpace>(nCells);
CES.B = cusp::array1d< ValueType, hostMemorySpace>(nCells);
//copy values from the lduMatrix to our equation system
thrust::copy(matrix().diag().begin(),matrix().diag().end(),CES.A.values.begin())
;//copy values of lduMatrix diag to A COO matrix
thrust::copy(matrix().lower().begin(),matrix().lower().end(),CES.A.values.begin()
+nCells);//copy values of lduMatrix lower to A COO matrix
thrust::copy(matrix().upper().begin(),matrix().upper().end(),CES.A.values.begin()
```

```
+nCells+NFaces);//copy values of lduMatrix upper to A COO matrix
//copy row and column indices of lower and upper to our equations system
//do not initialize the row and column values of diag to save time-> performed on GPU
thrust::copy(matrix().lduAddr().upperAddr().begin(),matrix().lduAddr().upperAddr().end(),
CES.A.row_indices.begin()+nCells);//copy row indices of lower into A COO matrix
thrust::copy(matrix().lduAddr().lowerAddr().begin(),matrix().lduAddr().lowerAddr().end(),
CES.A.column_indices.begin()+nCells);//copy column indices of lower into A COO matrix
thrust::copy(matrix().lduAddr().lowerAddr().begin(),matrix().lduAddr().lowerAddr().end(),
CES.A.row_indices.begin()+nCells+NFaces);//copy row indices of upper into A COO ma-
trix
thrust::copy(matrix().lduAddr().upperAddr().begin(),matrix().lduAddr().upperAddr().end(),
CES.A.column_indices.begin()+nCells+NFaces);//copy column indices of upper into A
COO matrix
thrust::copy(x.begin(),x.end(),CES.X.begin());//copy x of lower into x vector
thrust::copy(b.begin(),b.end(),CES.B.begin());//copy b of lower into b vector
//end COO, X, b initialize and fill
OFSP.iRes = -1;//initialization value for initial residual
OFSP.fRes = -1;//initialization value for final residual
OFSP.nIterations = 0;//initialization value for number of iterations
OFSP.minIter = minIter();//minimum iterations
OFSP.maxIter = maxIter();//maximum iterations
OFSP.relTol = relTolerance();//relative tolerance
OFSP.tol = tolerance();//tolerance
CFL_DiagPBiCGStab(&CES, &OFSP);//call the CUDA code to solve the system
thrust::copy(CES.X.begin(),CES.X.end(),x.begin());//copy the x vector back to Openfoam
solverPerf.initialResidual() = OFSP.iRes;//return initial residual
solverPerf.finalResidual() = OFSP.fRes;//return final residual
solverPerf.nIterations() = OFSP.nIterations;//return the number of iterations
//solverPerf.checkConvergence(tolerance(), relTolerance()) ; what should be passed here and
does this waste time?
// solverPerf.converged_=OFSP.converged;//private, cannot access
// solverPerf.singular_=OFSP.singular;//private, cannot access
return solverPerf;
}
```

Figura B8: Código C para el método de gradiente bi-conjugado estabilizado con precondicionador diagonal (*cufflink_DiagPBiCGStab*)

**Solucionador para los cálculos de matrices simétricas

```
#include "cufflink_CG.H"
//for COO and vector on host
```

```
#include <cusp/coo_matrix.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/copy.h>
#include "../CFL_Headers/cuspTypeDefs.H"
//end headers for COO and vectors
#include <iomanip>
#include <iostream>
#include <string>
#include <stdio.h>
//using namespace std;
#include "../CFL_Headers/OFSolverPerformance.H"
#include "../CFL_Headers/cusp_equation_system.H"
extern "C" void CFL_CG(cusp_equation_system *CES, OFSolverPerformance *OFSP);//extern
function in CUDA that solves the equation system
// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //
namespace Foam
{
defineTypeNameAndDebug(cufflink_CG, 0);
lduSolver::addsymMatrixConstructorToTable<cufflink_CG>
addcufflink_CGSymMatrixConstructorToTable_;
}
// * * * * * * * * * * * * * * * * Constructors * * * * * * * * * * * * * * //
Foam::cufflink_CG::cufflink_CG
(
const word& fieldName,
const lduMatrix& matrix,
const FieldField<Field, scalar>& coupleBouCoeffs,
const FieldField<Field, scalar>& coupleIntCoeffs,
const lduInterfaceFieldPtrsList& interfaces,
const dictionary& dict
)
:
lduSolver
(
```

```
fieldName,
matrix,
coupleBouCoeffs,
coupleIntCoeffs,
interfaces,
dict
)
{}
// * * * * * * * * * * * * * * * Member Functions * * * * * * * * * * * * * //
Foam::lduSolverPerformance Foam::cufflink_CG::solve
(
scalarField& x,
const scalarField& b,
const direction cmpt
) const
{
OFSolverPerformance OFSP;//container for solver performance
#include "../CFL_Headers/getGPUStorage.H"
// — Setup class containing solver performance data
lduSolverPerformance solverPerf("cufflink_CG",fieldName());
register label nCells = x.size();
register label NFaces = matrix().lower().size();
OFSP.nCells = nCells;
OFSP.nFaces = NFaces;
OFSP.debugCusp = false;
if (lduMatrix::debug >= 2) {
OFSP.debugCusp = true;
}
#include "../CFL_Headers/initializeCusp.H"
CFL_CG(&CES, &OFSP);//call the CUDA code to solve the system
thrust::copy(CES.X.begin(),CES.X.end(),x.begin());//copy the x vector back to Openfoam
```

```
solverPerf.initialResidual() = OFSP.iRes;//return initial residual
solverPerf.finalResidual() = OFSP.fRes;//return final residual
solverPerf.nIterations() = OFSP.nIterations;//return the number of iterations
//solverPerf.checkConvergence(tolerance(), relTolerance()) ; what should be passed here and
does this waste time?
return solverPerf;
}
```

Figura B9: Código C para el método de gradiente conjugado (*cufflink_DiagPBiCGStab*)