OPTIMIZACIÓN DEL MÉTODO DE RED DE VÓRTICES INESTACIONARIO PARA UNIDADES DE PROCESAMIENTO GRÁFICO

AGUSTÍN CURTO

Trabajo Especial de la Licenciatura en Ciencias de la Computación
Facultad de Matemática, Astronomía, Física y Computación
Universidad Nacional de Córdoba

Director: Lic. Carlos Bederián
Codirector: Doc. Sergio Preidikman

Diciembre 2020

«Optimización del Método de Red de Vórtices Inestacionario para Unidades de Procesamiento Gráfico» por Agustín Curto, se distribuye bajo una Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional

"La educación no cambia el mundo,
cambia a las personas que van a cambiar el mundo"

Paulo Freire

RESUMEN

Unsteady vortex-lattice method o método de red de vórtices inestacionario [4], UVLM de aquí en adelante, es un método numérico empleado para modelar la aerodinámica no lineal inestable de los vehículos micro aéreos de alas batientes, en dinámica de fluidos computacional. Se utiliza principalmente en las primeras etapas del diseño de aeronaves.

El *unsteady vortex-lattice method* [46], creado por el Dr. Preidikman junto con otros investigadores, es un método basado en el UVLM que amplía la capacidad de los UVLM convencionales, para permitir realizar un estudio aerodinámico de aleteo de alas en movimiento de desplazamiento, lo que implica una estela de deformación libre en el tiempo.

Aparte de este método, se desarrolló también una herramienta de simulación. El costo computacional de cada iteración de esta herramienta crece a medida que avanza la simulación. Como consecuencia, las simulaciones pequeñas toman algunos minutos en completarse mientras que, las de larga duración, se pueden extender a varias horas. Dicho costo limita la aplicabilidad de la herramienta.

En este trabajo final de la Licenciatura en Ciencias de la Computación, se paralelizó la herramienta existente con el objetivo de obtener resultados rápidos a bajo costo en simulaciones complejas. Al comienzo de este trabajo, una simulación típica tardaba poco más de 4:30 horas en ejecutarse. Al finalizar las optimizaciones, la misma simulación tarda menos de 7 minutos, lo que equivale a una ganancia de 40X.

Palabras claves: HPC, GPU, OMP, CUDA, UVLM

ABSTRACT

The unsteady vortex-lattice method [4], UVLM hereafter, is a numerical method used to model unsteady nonlinear aerodynamics of flapping-wings micro aerial vehicles in computational fluid dynamics. It is mainly used in the early stages of aircraft design.

The unsteady vortex-lattice method [46], which was created by Dr. Preidikman together with other researchers, is a method based on UVLM that extends the ability of conventional UVLMs to allow an aerodynamic study of flapping-wing in displacement motion, which involves a free-warping wake in time.

Apart from this method an ideal simulation tool was also developed. The computational cost of each iteration increases as the simulation progresses, making short simulations take some seconds to complete and extending long simulations for several hours. Such cost limits the tool applicability.

In this Bachelor of Computer Science final work, the existing tool was paralleled with the objective of getting fast results at low cost in more complex simulations. At the beginning, a typical simulation took over 4:30 hours to run. At the end of the optimizations, the same simulation takes less than 7 minutes, which equates to a profit of 40X.

Key words: HPC, GPU, OMP, CUDA, UVLM

AGRADECIMIENTOS

Quiero agradecer a todas las personas que me acompañaron a lo largo de mi carrera universitaria, desde mis familiares y amigos más cercanos, hasta quienes de una u otra forma, fueron parte de este proceso. Soy consciente de que nombrarlas a todas llevaría demasiado tiempo, es por esto que hago énfasis en las más cercanas.

A mis padres, Néstor y Alicia, quienes con su esfuerzo incansable me brindaron la posibilidad de ser estudiante a tiempo completo. Siempre me acompañaron y apoyaron en cada paso de este camino.

A mis hermanas Candela y Belén. Fueron una fuente de confianza y de aliento permanente, siempre estuvieron ahí cuando más las necesité.

A mi abuelo, Rumildo, quién hace poco dejó este mundo para irse a vivir una vida mejor, junto a mi abuela. No pude darle el regalo que tanto quería, por el cuál siempre me preguntaba con mucho orgullo: "¿Ya te recibís?", pero sé que desde algún lugar me está acompañando.

A mis directores, Charly y Sergio, por darme la oportunidad de realizar este trabajo final con ellos y así condecorar esta hermosa etapa de mi vida.

A la Universidad Pública, y especialmente a FaMAF y la UNC, por brindarme una educación de altísimo nivel. Formándome como persona y como profesional.

A mis amigos, Fran, Juan, Marco, Trucco y tantos otros compañeros y compañeras que, en mayor o menor medida, fueron parte de este proceso. Por todos los momentos compartidos, dentro y fuera de la facultad; por cada mate y cada tarde de estudio.

Por último, quiero agradecer a las tres instituciones que me brindaron los recursos computacionales para poder llevar a cabo este trabajo; ellas son CCAD y FaMAF, de la Universidad Nacional de Córdoba y Mercado Libre.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS

ACRÓNIMOS

UVLM Unsteady Vortex-Lattice Method

MAV Micro Air Vehicles

CFD Computational Fluid Dynamics

LEV Leading Edge Vortex

Div Divergense (función matemática)

FLOPS Floating Point Operations per Second

HPL High Performance Linpack

CPU Central Processing Unit

RAM Random Access Memory

DRAM Dynamic RAM

SRAM Static RAM

GPR General Purpose Register

FPR Floating Point Register

SPR Special Purpose Register

RPM Rotaciones por minuto

HDD Hard Drive Disk

SSD Solid State Disk

CUDA Compute Unified Device Architecture

GPGPU General Purpose computing on Graphics Processing Units

SM Streaming Multiprocessors

SIMT Single-Instruction, Multiple-Thread

API Application Programming Interface

RHS Right Hand Side

AI Arithmetic Intensity

TCP Texture Processing Clusters

1 INTRODUCCIÓN

1.1 MOTIVACIÓN

El Método de Red de Vórtices Inestacionario (UVLM) [4] es un método numérico utilizado en la dinámica de fluidos computacional, principalmente en las primeras etapas del diseño de aeronaves y en la educación aerodinámica a nivel universitario. El UVLM modela las superficies de sustentación, como un ala de una aeronave, como una lámina infinitamente delgada de vórtices discretos para calcular la elevación y la resistencia inducida. La influencia del espesor y la viscosidad se descuidan.

Los UVLM pueden calcular el flujo alrededor de un ala con una definición geométrica rudimentaria. Para un ala rectangular, es suficiente saber la envergadura (ancho del ala) y la cuerda (largo del ala). En el otro lado del espectro, pueden describir el flujo alrededor de una geometría de aeronave bastante compleja.

Al simular el campo de flujo, se puede extraer la distribución de fuerza alrededor del cuerpo simulado. Este conocimiento se utiliza para calcular los coeficientes aerodinámicos y sus derivados, los cuales son importantes para evaluar las cualidades de manejo de la aeronave en la fase de diseño conceptual. Con una estimación inicial de la distribución de presión en el ala, los diseñadores estructurales pueden comenzar a diseñar las superficies de sustentación.

Los detalles del Método de Red de Vórtices Inestacionario [46] creado por el Dr. Preidikman, junto con otros investigadores, serán explicados con detalle en el capítulo 2. Junto a este método, se desarrolló también una herramienta de simulación ideal para combinarse con dinámicas estructurales computacionales con el objetivo de proporcionar análisis aeroelásticos. El costo computacional de cada iteración de esta herramienta crece a medida que avanza la simulación. Como consecuencia, las simulaciones pequeñas toman algunos minutos en completarse mientras que, las de larga duración, se pueden extender a varias horas. Dicho costo limita la aplicabilidad de la herramienta.

Las unidades de procesamiento gráfico (GPU) son aceleradores con arquitecturas *manycore* y unidades de vectores anchas. Las GPU ponen un altísimo poder de cómputo y ancho de banda de memoria a disposición de aplicaciones con alto paralelismo. A diferencia de aceleradores anteriores, las GPU mantienen un costo asequible gracias a la aplicabilidad de las economías de escala en el mercado masivo de los videojuegos, lo cual da como resultado una relación costo/*performance* muy superior a la de los procesadores, cuando son aplicables a un problema.

Un *pipeline* GPGPU es un tipo de procesamiento paralelo entre una o más GPU y CPU, que analiza los datos como si estuvieran en una imagen u otra forma gráfica. Si bien las GPU funcionan a frecuencias

más bajas, generalmente disponen de una cantidad mayor de núcleos. Por lo tanto, las GPU pueden procesar muchas más imágenes y datos gráficos por segundo que una CPU tradicional. Migrar datos en forma gráfica y luego usar la GPU para escanear y analizarlos puede crear una gran aceleración.

Compute Unified Device Architecture, CUDA por sus siglas en inglés, hace referencia a una plataforma de computación en paralelo que incluye un compilador y un conjunto de herramientas de desarrollo creadas por *NVIDIA*, que permiten utilizar una variación del lenguaje de programación C para codificar algoritmos en GPU.

1.2 OBJETIVOS

Este trabajo final tiene como objectivo reducir el consumo de los recursos computacionales del modelo de simulación desarrollado por el Dr. Preidikman, mejorando la implementación ya existente mediante la utilización de técnicas y herramientas de computación de alto desempeño (HPC).

1.3 ESTRUCTURA

Esta tésis está estructurada de la siguiente manera:

- **Marco teórico**: explicación de los conceptos necesarios para entender, por un lado el modelo UVLM, y por otro, las técnicas de HPC.
- **Modelos computacionales: CPU vs GPU**: comparación entre las arquitecturas CPU y GPU, ventajas y desventajas de cada una.
- **Optimizaciones y resultados**: presentación de las distintas optimizaciones realizadas y un análisis de los resultados, para cada una de las optimizaciones.
- **Conclusiones y trabajo futuro**: resumen de todo el trabajo, posibles mejoras y trabajo futuro.

2

MARCO TEÓRICO

2.1 UVLM: MÉTODO DE RED DE VÓRTICES INESTACIONARIO

Desde hace varios años, la comunidad científica se ha centrado específicamente en el estudio de los insectos voladores y las aves pequeñas para inspirar el desarrollo de micro vehículos aéreos con alas batientes (*MAV*) [11], [18], [50], [17]. Sin embargo, todavía hay barreras técnicas importantes que superar, como comprender definitivamente cómo estas criaturas voladoras generan suficientes fuerzas aerodinámicas para impulsarse y mantenerse en el aire.

Desde un punto de vista numérico, claramente, el mejor enfoque para comprender el vuelo a pequeña escala sería resolver el flujo viscoso completo alrededor del insecto o ave. Sin embargo, las soluciones de las ecuaciones *Navier-Stokes* completas para campos de flujo inestables tridimensionales (3D), que tienen límites que experimentan movimientos complicados, relativamente grandes, son difíciles de resolver. Dificultades computacionales significativas y costos asociados con el uso de modelos basados en técnicas de dinámica de fluídos computacional (*CFD*) han llevado a la utilización de una gran variedad de modelos aerodinámicos para estudiar el vuelo natural [51], [44], [12], [1], [2], [26].

Actualmente, el uso de métodos de red de vórtices inestacionarios (UVLM) ha ido ganando terreno en el estudio de problemas no estacionarios, en los que los métodos de estela libre se convierten en una necesidad debido a la complejidad geométrica [4], [45], [3], [30], [27], [28].

En relación con la aerodinámica del ala de aleteo, se pueden identificar cinco términos como los principales contribuyentes a las cantidades de flujo durante el vuelo estacionario. Incluyen los efectos debidos a la traslación y rotación del ala, el borde de vórtice principal (*LEV*), la captura de estela, la viscosidad y la masa añadida. Los UVLM capturan la mayoría, salvo los efectos viscosos y LEV. Los efectos viscosos para el rango de números de *Reynolds* (75-4000) de MAVs/insectos flotantes pueden ser descuidados [11], [12], lo que hace que el uso de UVLM sea adecuado para el estudio de la aerodinámica del ala de aleteo.

En este nuevo modelo, desarrollado por el Dr. Preidikman [46], se amplió significativamente la capacidad de los UVLM para estudiar la aerodinámica de una mosca de la fruta (*drosophila melanogaster*) al incluir:

- separación de borde de ataque
- la estructura del cuerpo del insecto (cabeza, tórax y abdomen)
- diferentes patrones cinemáticos

El presente modelo aerodinámico tiene en cuenta todas las posibles interferencias aerodinámicas y permite predecir:

- el campo de flujo alrededor del cuerpo y las alas de un insecto
- la distribución de vorticidad espacial y temporal unida al cuerpo y las alas del insecto
- la distribución de vorticidad en las estelas emitidas por los bordes afilados de las alas
- la posición y forma de estas alas
- las cargas aerodinámicas inestables que actúan sobre las alas

Un estudio aerodinámico de aleteo de alas en movimiento de desplazamiento, por medio de un UVLM, que implica una estela de deformación libre en el dominio del tiempo, geometrías dependientes del tiempo y flujos en gran parte adjuntos, no está disponible en la literatura, y es enfoque de este nuevo modelo denominado: **Método de Red de Vórtices Inestacionarios Modificado**.

2.1.1 *Modelo del Insecto*

El modelo de insecto adoptado en este trabajo para estudiar la aerodinámica de las alas batientes, corresponde a una mosca de la fruta (*drosophila melanogaster*). Dicho modelo se basa en el trabajo de *Markow* y *O'Grady* [34] para preservar ciertos parámetros morfológicos como la longitud del ala R, la longitud del cuerpo L_b, la longitud máxima de la cuerda c_{max} y la geometría del ala y el cuerpo del insecto.

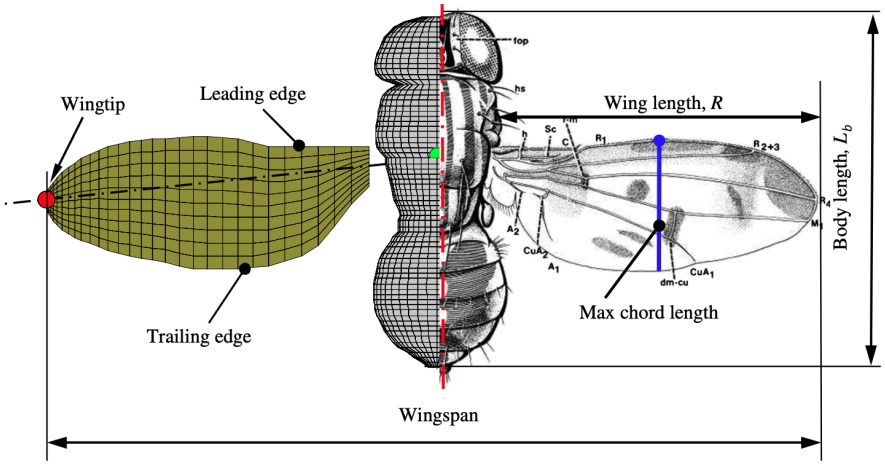

Figura 1: Modelo geométrico de un insecto y las definiciones de parámetros morfológicos [46].

Para simplificar, cada parte del cuerpo del insecto (cabeza, tórax y abdomen) fue modelada como una superficie de revolución. Las superficies de revolución que definen el cuerpo del insecto, así como

las superficies que modelan las alas del insecto, se discretizaron utilizando elementos cuadrilaterales de cuatro nodos simples, no planos. Esta discretización será explicada más adelante en 2.1.3.2.

2.1.2 *Modelo Cinemático*

Para describir la trayectoria de cualquier punto arbitrario en el ala del insecto, se utilizaron cuatro sistemas de referencia:

a) un inercial o sistema de referencia newtoniano N

b) un sistema fijo al cuerpo T, ubicado en el centro de masa del tórax

c) un sistema de referencia fijado al trazo plano Z

d) dos sistemas de referencia fijados a cada raíz del ala, en orden para facilitar su discretización especial, B para el ala izquierda y A para el ala derecha

Se puede observar estos sistemas en la siguiente figura.

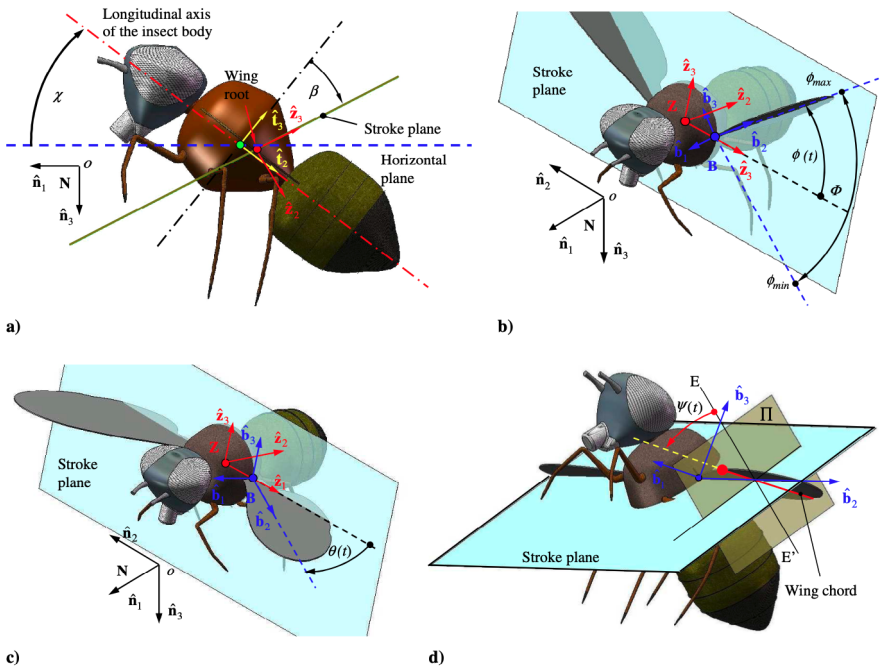

Figura 2: Definiciones de parámetros de carrera: a) ángulo del cuerpo y ángulo del plano de carrera, b) ángulos de posición de carrera, c) ángulo de desviación de carrera, y d) ángulo de rotación [46].

La orientación del cuerpo del insecto se ve afectada exclusivamente por un cambio en el ángulo del cuerpo χ y se obtiene mediante una rotación $(1-\texttt{rotation})$ del marco de referencia T.

La orientación del plano de trazo con respecto al marco inercial N se logra en dos etapas. Primero, se coloca el plano de trazo perpendicularmente al eje longitudinal del insecto mediante el ángulo del cuerpo, y luego, el plano de trazo está orientado con respecto a un eje

perpendicular al eje longitudinal de la criatura, por medio del ángulo del plano de carrera β.

La orientación del ala en relación con el plano de trazo se define por tres ángulos:

- el ángulo de posición de carrera $\phi(t)$
- el ángulo de desviación de carrera $\theta(t)$
- el ángulo de rotación alrededor del ala eje longitudinal, $\psi(t)$

Se define la orientación del ala con la secuencia de rotaciones (1-3-2) dada por los ángulos de *Euler* $\phi(t)$, $\theta(t)$ y $\psi(t)$ respectivamente.

El ángulo de posición del trazo está formado por la proyección del eje longitudinal del ala en el plano de carrera y el vector unitario $\hat{z_1}$ y es positivo cuando el ala está en la posición ventral. El ángulo de desviación del trazo se define como el ángulo formado por el eje longitudinal del ala y el plano de carrera, y se considera positivo cuando las alas están por encima del plano de carrera. El ángulo de rotación se mide en un plano $\prod$, que tiene una orientación en el espacio tridimensional, que siempre es normal al vector unitario $\hat{b_2}$ fijado al ala; se define como el ángulo formado por la cuerda del ala y la recta línea EE', que se fija al plano $\prod$ y coincide con la dirección del vector unitario $\hat{b_1}$ en $t = 0$. Este ángulo es positivo en la carrera descendente.

2.1.3 *Modelo Aerodinámico*

El método ampliado se puede aplicar a los flujos de elevación y no elevación 3D. Es general, en el sentido de que la superficie del cuerpo puede sufrir una deformación dependiente del tiempo, mientras ejecuta cualquier tipo de maniobra en el espacio rodeado de aire en movimiento. Se supone que el flujo alrededor del cuerpo (es decir, el cuerpo y las alas del insecto) es irritable e incompresible en todo el campo de flujo, excepto al lado de los límites sólidos del cuerpo y en la estela. Este enfoque permite considerar efectos aerodinámicos no lineales e inestables, asociados con grandes ángulos de ataque y deformaciones estáticas. El UVLM también permite tener en cuenta todas las posibles interferencias aerodinámicas, así como estimar la distribución de vorticidad espacial y temporal unida a la superficie del cuerpo, la distribución de vorticidad en el cuerpo, así como la posición y la forma de las estelas que se desprenden de los agudos bordes de las alas.

Como resultado del movimiento relativo entre el cuerpo y el fluído, se genera vorticidad en el fluído en una región delgada adyacente a la superficie del cuerpo (la capa límite). Esta vorticidad se desprende de los bordes afilados y forma la estela. Se considera que tanto las capas del límite como las estelas son hojas de vorticidad.

La hoja de vórtice encuadernado representa la capa límite en la superficie del cuerpo, y se especifica su posición (es decir, se adhiere y se mueve con el cuerpo, no con las partículas de fluído). Para el

caso de alas delgadas, las hojas de vórtice en las superficies superior e inferior se fusionan en una sola superficie, a lo largo de la línea de curvatura. Por otro lado, las posiciones de las hojas de vórtice libre, que representan las estelas, no se especifican a priori; se les permite deformarse libremente, hasta que asuman posiciones libres de fuerza según lo determine la solución. Los dos tipos de hojas de vórtice se unen a lo largo de los bordes afilados donde la separación ocurre; los mismos bordes en los que se impone la condición de *Kutta* en un flujo constante.

Existe una relación cinemática entre vorticidad y velocidad, tal que, si hay vorticidad en cualquier parte del campo de flujo, entonces hay velocidad asociada a ella en todas partes en el campo de flujo; la velocidad decae con la distancia de la vorticidad. La vorticidad en la estela, en cualquier momento dado, se generó y se desprendió de las alas en un momento anterior; la velocidad asociada con esta vorticidad de estela afecta el flujo cerca del ala y, por lo tanto, las cargas en las alas. Como resultado, las cargas aerodinámicas dependen de la historia del movimiento, y la estela es el "historiador". A medida que la vorticidad en la estela se transporta hacia abajo, su influencia disminuye y, por lo tanto, la memoria del historiador se desvanece. El desvanecimiento de la memoria es algo muy bueno, ya que significa que la parte de la estela que debe tenerse en cuenta no tiene por qué ser muy larga.

Ramamurti y *Sandberg* [44] mostraron que los efectos de la viscosidad, en flujo inestable que rodea un ala tridimensional de una *drosophila* en proceso de movimiento de aleteo, son mínimos y que las fuerzas de elevación y arrastre están dominadas por los efectos de inercia. La cuerda, basada en el número de *Reynolds*, que caracteriza el vuelo de insectos, es relativamente baja, y entonces la pregunta surge naturalmente: ¿se puede usar UVLM de manera confiable para predecir las cargas aerodinámicas en alas batientes? La respuesta se desarrollará a continuación.

2.1.3.1 *Formulación matemática*

La ecuación de continuidad para flujo incompresible gobierna el campo de velocidad $\mathbf{V}(\mathbf{r}, t)$:

$$\mathrm{Div}\mathbf{V}(\mathbf{r}, t) = 0 \tag{1}$$

La dependencia del tiempo es introducida por el límite móvil. El campo de vorticidad $\boldsymbol{\Omega}$ y el campo de velocidad $\mathbf{V}$ coexisten y están cinemáticamente relacionados:

$$\boldsymbol{\Omega} = \nabla \times \mathbf{V}(\mathbf{r}, t) \tag{2}$$

De esta relación se deduce que la velocidad asociada con un segmento recto y finito de una línea de vórtice con circulación $\Gamma(t)$, está dada por la ley de *Biot-Savart*:

$$\mathbf{V}(\mathbf{r}, t) = \frac{\Gamma(t)}{4\pi} \frac{\boldsymbol{\omega} \times \mathbf{r}_1}{\| \boldsymbol{\omega} \times \mathbf{r}_2 \|_2^2} [\boldsymbol{\omega} \cdot (\hat{\mathbf{e}}_1 - \hat{\mathbf{e}}_2)] \tag{3}$$

Aquí, $\mathbf{r}$ es el punto de campo donde se calcula la velocidad, $\mathbf{r}_1$ y $\mathbf{r}_2$ son los vectores de posición, dependientes del tiempo del punto de campo en relación con los extremos del segmento de vórtice recto, $\hat{\mathbf{e}}_1$ y $\hat{\mathbf{e}}_2$ son vectores unitarios paralelos a $\mathbf{r}_1$ y $\mathbf{r}_2$, y $\boldsymbol{\omega} = \mathbf{r}_1 - \mathbf{r}_2$. La velocidad dada por la ecuación (3) satisface la ecuación (1) y es irrotacional ($\boldsymbol{\Omega} = 0$) en todas partes, excepto en el segmento de vórtice.

Para un punto de campo en el segmento de vórtice, o muy cerca del mismo o su extensión, $\boldsymbol{\omega}$ está cerca o es casi paralela a $\mathbf{r}_1$. Esto provoca el comportamiento problemático de $\mathbf{V}(\mathbf{r}, t)$, en la ecuación (3). Dicho comportamiento se puede eludir fácilmente introduciendo un "radio de corte" δ en la ecuación (3):

$$\mathbf{V}(\mathbf{r}, t) = \frac{\Gamma(t)}{4\pi} \frac{\boldsymbol{\omega} \times \mathbf{r}_1}{\| \boldsymbol{\omega} \times \mathbf{r}_2 \|_2^2 + (\delta \| \boldsymbol{\omega} \|_2)^2} [\boldsymbol{\omega} \cdot (\hat{\mathbf{e}}_1 - \hat{\mathbf{e}}_2)] \tag{4}$$

2.1.3.2 *Discretización de las hojas de vórtice*

Las hojas de vórtice encuadernado de un UVLM son reemplazadas con una red de segmentos de vórtice cortos y rectos con constante circulación. Estos segmentos dividen la superficie del cuerpo y las alas del insecto en elementos de área, que en general son no planos, con segmentos discretos de vórtice a lo largo de los bordes. El modelo se completa uniendo líneas de vórtice libre, que representan las hojas de vórtice libre (estelas), a la red de vórtices encuadernado a lo largo de los bordes donde se produce la separación, como los bordes de salida y los bordes de ataque de las alas. Las ubicaciones en las que se produce la separación son de entrada y no están determinadas por la solución. Sin embargo, las redes de vórtice representando las estelas (las posiciones de los segmentos de vórtice y las circulaciones a su alrededor) se determinan como parte de la solución.

La figura 3 muestra ejemplos de mallas para las hojas de vórtice encuadernado. En ambos casos, existe una brecha entre la raíz del ala y las zonas de separación (zona 1 para el borde de ataque y zona 2 para el borde de salida). Este ajuste mejora notablemente la forma de la estela cerca de la raíz del ala.

El campo de velocidad asociado con la perturbación creada por el cuerpo en movimiento, es la superposición de los campos asociados con la vorticidad en la red ligada, en la superficie del cuerpo en movimiento y en las estelas que se deforman libremente.

2.1.3.3 *Condiciones de borde*

La ecuación gobernante del problema se complementa con las siguientes condiciones de contorno:

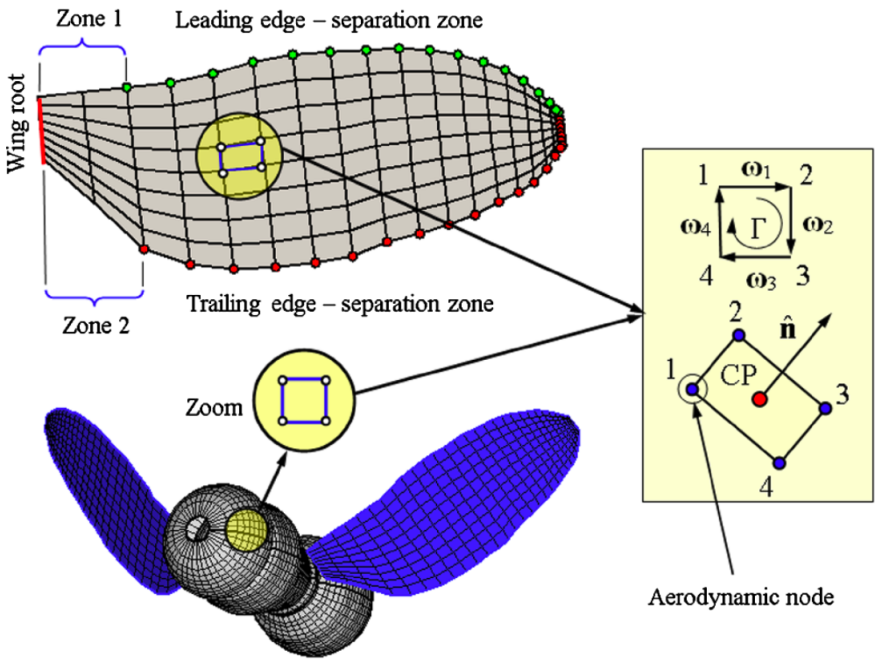

Figura 3: Discretización de hojas de vórtice encuadernado que representan el cuerpo y las alas de un insecto [46].

- La regularidad en la condición de infinito, requiere que la velocidad del campo asociado con la interrupción se deteriore fuera del cuerpo y su estela. Por lo tanto:

$$\lim_{\|\mathbf{r}\|_2 \to \infty} \|\mathbf{V}_B(\mathbf{r}, t) + \mathbf{V}_W(\mathbf{r}, t) + \mathbf{V}_{SW}(\mathbf{r}, t)\| = 0 \tag{5}$$

  donde $\|\mathbf{r}\|_2$ es la distancia del cuerpo y sus estelas, $\mathbf{V}_B(\mathbf{r}, t)$ es la velocidad asociada con la red de vórtices encuadernados, $\mathbf{V}_W(\mathbf{r}, t)$ es la velocidad asociada con la red de vórtice libre, que se desprende del borde posterior y la punta del ala, y $\mathbf{V}_{SW}(\mathbf{r}, t)$ es la velocidad asociada con el enrejado de vórtice libre, que se desprende de los principales bordes del ala. El campo de velocidad obtenido de la ecuación (4) satisface esta condición.

- La condición de no penetración requiere que, sobre toda la superficie del cuerpo y las alas del insecto, la componente normal de la velocidad del fluído, en relación con la superficie del cuerpo, debe ser cero:

$$(\mathbf{V}_\infty + \mathbf{V}_B + \mathbf{V}_W + \mathbf{V}_{SW} - \mathbf{V}_p) \cdot \hat{\mathbf{n}} = 0 \tag{6}$$

  Debido a que las hojas de vórtice se reemplazan por redes de vórtice, la condición de no penetración dada por la ecuación (6) solo se satisface en un punto en cada panel; estos se denominan **puntos de control** (CP) y están ubicados en el centroide de las esquinas de cada panel (ver figura 3).

Además de las condiciones de contorno, existen las siguientes tres condiciones:

- Debe haber una presión continua en la estela. Para un fluído invisible, el teorema de *Kelvin-Helmholtz* requiere que la vorticidad sea transportada con las partículas fluidas. Esta condición se usa para obtener las posiciones de los segmentos de vórtice que comprenden las redes, representando la estela.
- Debe haber conservación espacial de la circulación: el campo de vorticidad es divergente. Esta condición es satisfecha por considerar que las redes de vórtices están compuestas de bucles cerrados de segmentos de vórtice con la misma circulación.
- Se debe cumplir la condición inestable de *Kutta*. Las presiones en las superficies superior e inferior deben desaparecer a lo largo de los bordes donde se produce la separación; esto requiere que toda la vorticidad generada a lo largo de estos bordes se desprenda y, por lo tanto, esta condición determina la resistencia de la vorticidad en la estela.

2.1.3.4 *Modelo de separación de bordes de ataque*

Debido a que la cinemática de los insectos con alas es compleja, el desprendimiento de vórtices desde el borde de ataque, depende del ángulo entre la velocidad del fluído local y el plano del ala (ángulo de ataque efectivo). Varios trabajos en separación de bordes de ataque de alas de aviones convencionales, han informado que el flujo unido al ala comienza a separarse cuando el ángulo de ataque excede un valor crítico de 12-15 grados [28]. *Dickinson y Götz* [11], encontraron que a 9 grados (un umbral muy por debajo de los utilizados por los insectos) una delgada burbuja de separación, apenas visible en imágenes de video, se forma rápidamente en la superficie superior del perfil aerodinámico y permanece estable durante todo el desplazamiento.

Varios autores han desarrollado herramientas numéricas basadas en métodos de red de vórtice [35], [32], [47], que explican el borde de ataque en el barrido de las alas delta. En el modelo del presente trabajo, se modificó y extendió un UVLM existente para incluir los efectos de la separación del borde de ataque y luego se usó la versión modificada para calcular las cargas aerodinámicas en aleteo de alas. Como en el método de red vorticial, el sistema *LEV* también está representado por una familia de líneas discretas de vórtice, y el campo de velocidad asociado con el borde del sistema de ataque se calcula con la ley de *Biot-Savart*. Este campo de flujo se agrega a los generados por las otras redes y la corriente libre. Los bordes de ataque se incluyeron mediante un esquema basado en un mecanismo de encendido/apagado. Este mecanismo consiste principalmente en computar el valor del ángulo efectivo de ataque α_e en cada paso de tiempo, comparándolo con un valor de referencia; en los presentes ejemplos, la referencia es $\alpha_c = 12$ grados. Si $\alpha_e \geqslant \alpha_c$, la separación del borde de ataque está incluída; por el contrario, si $\alpha_e < \alpha_c$, se omite la separación del borde de ataque.

Una vez que un segmento de vórtice se introduce en la estela, siempre permanece en la misma. En la figura 4, se presenta la definición del ángulo α_e, así como las estelas que se desprenden tanto del borde posterior como del borde delantero.

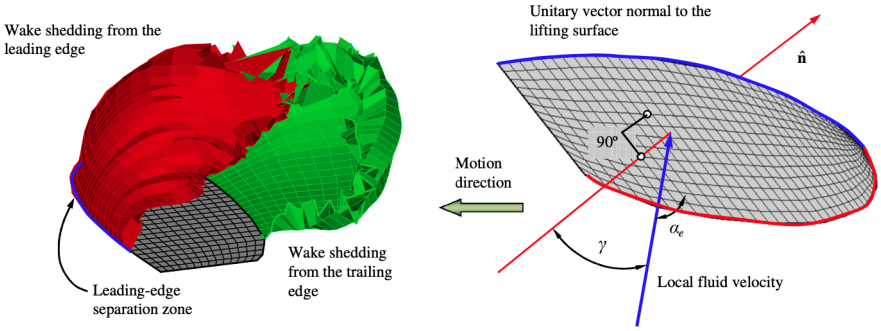

Figura 4: Zonas de separación y definición del ángulo α_e [46].

2.1.3.5 *Coeficientes de influencia aerodinámica*

Generalmente, el componente normal de la velocidad de una partícula fluida, relativa a un punto de control, depende de la superposición de la velocidad de los campos asociados con:

- las redes reticuladas de vórtice
- la red de vórtice libre
- la corriente libre

El componente normal de la velocidad en el punto de control del i-ésimo panel, asociado con el ciclo cerrado de segmentos de vórtice con circulaciones unitarias a lo largo de los bordes del panel j-ésimo, es denotado por a_{ij}. En consecuencia, el componente normal de la velocidad en el punto de control i, asociado con todos los vórtices enlazados, viene dado por $\sum_{j=1}^{N} a_{ij}\Gamma_j$, donde N es el número de paneles en las redes enlazadas y Γ_j es la magnitud de la circulación alrededor del circuito cerrado de segmentos de vórtice, a lo largo de los bordes del panel j. La condición de no penetración para el i-ésimo panel se puede escribir de la siguiente manera:

$$\sum_{j=1}^{N} a_{ij}\Gamma_j + (\mathbf{V}_\infty + \mathbf{V}_W + \mathbf{V}_{SW} - \mathbf{V}_p)\cdot\hat{\mathbf{n}} = 0 \tag{7}$$

donde $\mathbf{V}_p$ es la velocidad de la superficie del cuerpo y $\hat{\mathbf{n}}_i$ es el vector unitario normal a la superficie, en el punto de control del i-ésimo panel. La ecuación (7) debe satisfacerse simultáneamente en los puntos de control de todos los paneles, es decir, para $i = 1 \cdots N$. Los campos de velocidad asociados con la vorticidad en las estelas,

la velocidad de flujo libre y la velocidad debida a la cinemática del cuerpo ya se conocen y puede transferirse al lado derecho (RHS):

$$\mathrm{RHS}_i = -(\mathbf{V}_\infty + \mathbf{V}_W + \mathbf{V}_{SW} - \mathbf{V}_p)\cdot \hat{\mathbf{n}}_i \tag{8}$$

Luego, escribiendo la ecuación (7) para cada panel en la red de vórtice encuadernado, se obtiene el siguiente sistema de ecuaciones lineales:

$$\mathbf{A}(t)\boldsymbol{\Gamma}(t) = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{N2} & \cdots & A_{NN} \end{bmatrix} \begin{Bmatrix} \Gamma_1 \\ \Gamma_2 \\ \vdots \\ \Gamma_N \end{Bmatrix} = \begin{Bmatrix} \mathrm{RHS}_1 \\ \mathrm{RHS}_2 \\ \vdots \\ \mathrm{RHS}_N \end{Bmatrix} \tag{9}$$

Si las diferentes partes del insecto (cabeza, tórax, abdomen y alas) no se mueven entre sí, entonces los coeficientes de influencia son evaluados solo una vez; de lo contrario, se vuelven a evaluar en cada paso. La cabeza, el tórax y el abdomen se modelan como solo cuerpo rígido, y las alas tienen un movimiento prescrito en todo el ciclo de carrera. Por lo tanto, las únicas partes de influencia de la matriz aerodinámica $\mathbf{A}(t)$, que se actualizarán en cada paso, son aquellos que tienen en cuenta la interferencia aerodinámica del cuerpo-ala y el ala-ala. Una vez que la ecuación (9) está resuelta, el siguiente paso es calcular las cargas aerodinámicas.

2.1.3.6 *Cargas aerodinámicas*

Las cargas aerodinámicas en las superficies de elevación (alas de insecto) se calculan de la siguiente manera:

- Para cada elemento, el salto de presión en el punto de control, es calculado con la ecuación inestable de *Bernoulli*

  $$\frac{\partial}{\partial t}\boldsymbol{\Psi}(\mathbf{r},t) + \frac{1}{2}\mathbf{V}(\mathbf{r},t)\cdot\mathbf{V}(\mathbf{r},t) + \frac{p(\mathbf{r},t)}{\rho} = \frac{1}{2}\mathbf{V}_\infty\cdot\mathbf{V}_\infty + \frac{p_\infty}{\rho} \tag{10}$$

  donde $\frac{\partial}{\partial t}$ denota la derivada de tiempo parcial, en una ubicación fija en un marco de referencia inercial.

- La fuerza sobre cada elemento se calcula como el producto del salto de presión, multiplicado por el área del elemento. Esta área se obtiene mediante la suma de la mitad de los productos cruzados de dos vectores, a lo largo de los bordes adyacentes del panel que multiplica por el vector unitario normal, obtenido del producto cruzado de las dos diagonales.

- Las fuerzas y momentos resultantes se calculan como el vector suma de las fuerzas y sus momentos sobre un punto común.

Sin embargo, como el UVLM se basa en la teoría del perfil aerodinámico delgado, no cuenta la succión de vanguardia [33], y solo

el componente normal no circulatorio retiene la velocidad, es decir, la contribución de presión al ascensor local. La contribución de las fuerzas en los elementos de la red al arrastre/empuje inducido, está alineado con la velocidad instantánea no circulatoria, y se puede calcular por medio de múltiples métodos [28], [48], [15].

Para calcular con precisión el empuje generado por el movimiento de aleteo de un ala plana, la contribución de la fuerza de succión de vanguardia debe incluirse en los cálculos, haciendo que el vector de fuerza aerodinámica resultante se incline hacia el borde de ataque. El cálculo de esta fuerza no se ha incluido en el presente modelo, lo que lleva a una subestimación del empuje total.

En su forma actual, la evaluación de $\frac{\partial}{\partial t}\mathbf{\Psi}(\mathbf{r}, t)$ es problemática, pero este término puede expresarse de manera que su evaluación sea relativamente fácil [29], [42], [43]. Una vez que se han calculado las cargas, los paneles en las estelas son "convocados" a sus nuevas posiciones [25]:

$$\mathbf{R}_{node}(t + \Delta t) \approx \mathbf{R}_{node}(t) + \mathbf{V}_{node}(t)\Delta t \tag{11}$$

donde Δt es el incremento de tiempo.

Como todas estas cantidades son funciones del tiempo, la cuestión de qué cantidades instantáneas utilizar en la aproximación, se eleva. Hay varias opciones; por ejemplo, uno puede usar las cantidades que se calcularon en el paso de tiempo anterior, el paso de tiempo actual o sus valores promedio para los dos pasos de tiempo. En los útimos dos casos, se necesitan iteraciones que aumentan el tiempo de cálculo. *Kandil* [25] mostró que la primera opción es estable, y hay pequeñas diferencias en los resultados calculados para las diversas opciones; por lo tanto, la primera opción se utilizó para calcular todos los resultados en este trabajo.

Luego se repiten los pasos anteriores para encontrar las cargas en el siguiente paso.

2.2 HPC: HIGH PERFORMANCE COMPUTING

2.2.1 *Definición*

High Performance Computing (HPC) es una colección de múltiples disciplinas interrelacionadas. Es un campo de trabajo que se relaciona con todas las facetas de la tecnología, la metodología y la aplicación, asociadas con el logro de la mayor capacidad computacional posible en cualquier momento y tecnología. Involucra a una clase de máquinas denominadas **supercomputadoras** para realizar una amplia gama de problemas computacionales lo más rápido posible. La acción de realizar una aplicación en una supercomputadora se denomina *supercomputing* y es sinónimo de HPC.

2.2.2 *Aplicación*

El propósito del HPC es obtener respuestas a preguntas que no pueden abordarse adecuadamente por sí solas a través del empirismo, la teoría o incluso computadoras personales. Históricamente, las supercomputadoras se han aplicado a la ciencia y la ingeniería, pero la gama de problemas que las supercomputadoras pueden abordar se extiende más allá de los estudios científicos y de ingeniería clásicos para incluir desafíos en socioeconomía, gestión y aprendizaje de *big data*, control de procesos y seguridad nacional.

2.2.3 *Rendimiento y métricas*

Si bien la noción de rendimiento puede ser intuitiva, no es simple. Una métrica es un parámetro operacional observable y cuantificable de una supercomputadora. La métrica más utilizada es *floating point operations per second* (FLOPS). Una operación de punto flotante es una suma o multiplicación de dos números reales o de punto flotante. A partir de junio de 2019, todos los sistemas TOP500 ofrecen un petaflops o más, con el nivel de entrada a la lista ahora en 1.32 petaflops.

Hay dos razones importantes para utilizar una computadora en paralelo: tener acceso a más memoria u obtener un mayor rendimiento. Es fácil caracterizar la ganancia de memoria, ya que la memoria total es la suma de las memorias individuales. La velocidad de una computadora en paralelo es más difícil de caracterizar. Algunas medidas teóricas [16] para expresar y juzgar la ganancia en la velocidad de ejecución al pasar a una arquitectura paralela se discutirán a continuación. Un enfoque simple para definir la aceleración es dejar que el mismo programa se ejecute en un solo procesador y en una máquina paralela con p procesadores, y luego comparar tiempos de ejecución. Se utiliza T_1 para hacer referencia al tiempo de ejecución en un solo procesador y T_p para el tiempo en p procesadores. La aceleración se define como:

$$S_p = \frac{T_1}{T_p}$$

En el caso ideal, $T_p = \frac{T_1}{p}$, pero en la práctica $S_p \leqslant p$. Para medir qué tan lejos se encuentra un programa de la aceleración ideal, se introduce la eficiencia $E_p = \frac{S_p}{p}$, donde claramente $0 \leqslant E_p \leqslant 1$.

Hay varias razones por las que la velocidad real es menor que p. Por un lado, el uso de más de un procesador requiere comunicación, que es una sobrecarga que no formaba parte del cálculo original. En segundo lugar, si los procesadores no tienen exactamente la misma cantidad de trabajo que hacer, pueden estar inactivos parte del tiempo (esto se conoce como desequilibrio de carga), lo que nuevamente reduce la aceleración realmente alcanzada. Finalmente, el código puede tener secciones que son inherentemente secuenciales. La comunicación entre procesadores es una fuente importante de pérdida de eficiencia.

Claramente, un problema que se puede resolver sin comunicación será muy eficiente. Estos problemas, que en realidad consisten en una serie de cálculos completamente independientes, se denominan vergonzosamente paralelos; tendrán una velocidad y eficiencia casi perfectas.

En ocasiones, dividir un problema entre muchos procesadores no tiene sentido: en cierto punto, simplemente no hay suficiente trabajo para que cada procesador funcione de manera eficiente. En cambio, en la práctica, los usuarios de un código paralelo elegirán el número de procesadores para que coincida con el tamaño del problema, o resolverán una serie de problemas cada vez mayores en un número de procesadores correspondientemente creciente. En ambos casos es difícil hablar de aceleración. En cambio, se utiliza el concepto de escalabilidad. Existen dos tipos de escalabilidad. La llamada **escalabilidad fuerte**, es en efecto lo mismo que la aceleración, discutida anteriormente. Se dice que un programa muestra una gran escalabilidad si, divididiendolo en más procesadores, muestra una aceleración perfecta o casi perfecta, es decir, el tiempo de ejecución disminuye linealmente con el número de procesadores. En términos de eficiencia, esto se puede describir como:

$$\left.\begin{array}{l} N \equiv \text{ constant} \\ P \to \infty \end{array}\right\} \Rightarrow E_p \approx \text{ constant}$$

donde N es el tamaño del problema, mientras que P es la cantidad de procesadores.

Más interesante aún, la **escalabilidad débil** es un término definido de manera más vaga. Describe el comportamiento de ejecución, a medida que el tamaño del problema y el número de procesadores crecen, pero de tal manera que la cantidad de datos por procesador se mantiene constante. Medidas como la aceleración son algo difíciles de informar, ya que la relación entre el número de operaciones y la cantidad de datos puede ser complicada. Si esta relación es lineal, se podría afirmar que la cantidad de datos por procesador se mantiene constante e informar que el tiempo de ejecución en paralelo es constante a medida que crece el número de procesadores.

En términos de eficiencia:

$$\left.\begin{array}{l} N \to \infty \\ P \to \infty \\ M = \frac{N}{P} \equiv \text{ constant} \end{array}\right\} \Rightarrow E_p \approx \text{ constant}$$

Debido a que existen literalmente millones de problemas en cuales aplicar HPC, la comunidad selecciona problemas específicos, alrededor de los cuales estandarizar. Dichos programas de aplicación estandarizados son puntos de referencia. Un punto de referencia de *supercomputing* particularmente utilizado es el **High Performance Linpack** (HPL) [13], el cual resuelve un conjunto de ecuaciones lineales en forma de matriz densa. Un punto de referencia proporciona un

medio de evaluación comparativa entre dos sistemas independientes, midiendo sus tiempos respectivos para realizar el mismo cálculo. La comunidad de HPC ha seleccionado HPL como un medio para clasificar las supercomputadoras, como lo representa la Lista de las 500 principales, que comenzó en 1993 (figura 5). Aunque también se emplean otros puntos de referencia para enfatizar ciertos aspectos de una supercomputadora o representar una cierta clase de programas.

Recientemente un nuevo punto de referencia ha surgido para las mediciones en TOP500. EL nuevo benchmark HPL-AI busca resaltar la convergencia emergente de cargas de trabajo de computación de alto rendimiento (HPC) e inteligencia artificial (AI). Mientras que la HPC tradicional se centró en ejecuciones de simulación para modelar fenómenos en física, química, biología, etc., los modelos matemáticos que impulsan estos cálculos requieren, en su mayor parte, precisión de 64 bits. Por otro lado, los métodos de aprendizaje automático que impulsan los avances en IA logran los resultados deseados en formatos de precisión de 32 bits e incluso de punto flotante más bajos. Esta menor demanda de precisión impulsó un resurgimiento del interés en nuevas plataformas de hardware que ofrecen una combinación de niveles de rendimiento y ahorros de energía sin precedentes para lograr la clasificación y la fidelidad de reconocimiento que ofrecen los formatos de mayor precisión.

La 56 edición del TOP500 vio a la supercomputadora japonesa **Fugaku** solidificar su estado número uno, aumentando su rendimiento en **HPL** de 416 a 442 petaflops. Más significativamente, Fugaku aumentó su rendimiento en el nuevo punto de referencia **HPC-AI** de precisión mixta a 2.0 exaflops, superando su propia marca de 1.4 exaflops registrada seis meses antes. Estos resultados son importantes, ya que representan las primeras medidas de referencia por encima de un exaflop para cualquier precisión en cualquier tipo de hardware.

2.2.4 *Sistemas de HPC*

El aspecto más visible del campo de HPC son las computadoras de alto rendimiento, o simplemente supercomputadoras, en sí mismas. Hoy en día, estas máquinas aparecen como filas y filas de muchos bastidores que ocupan miles de pies cuadrados y consumen potencialmente varios megavatios de energía eléctrica. Estar en presencia de una (que a menudo significa estar literalmente dentro de ella) ofrece una experiencia completamente diferente en términos de ruido, gradientes de temperatura que cambian rápidamente y muchas luces parpadeantes. Incluso el observador más serio no puede evitar quedar asombrado por la impresionante masividad de tales sistemas, la ingeniería mediante la cual se logran, el compromiso que representan con los límites de la capacidad informática y los problemas que solo ellos pueden resolver.

El despliegue de una supercomputadora de última generación es verdaderamente una importante empresa de ingeniería que implica tiempo, gastos y experiencia, así como gestión y mantenimiento

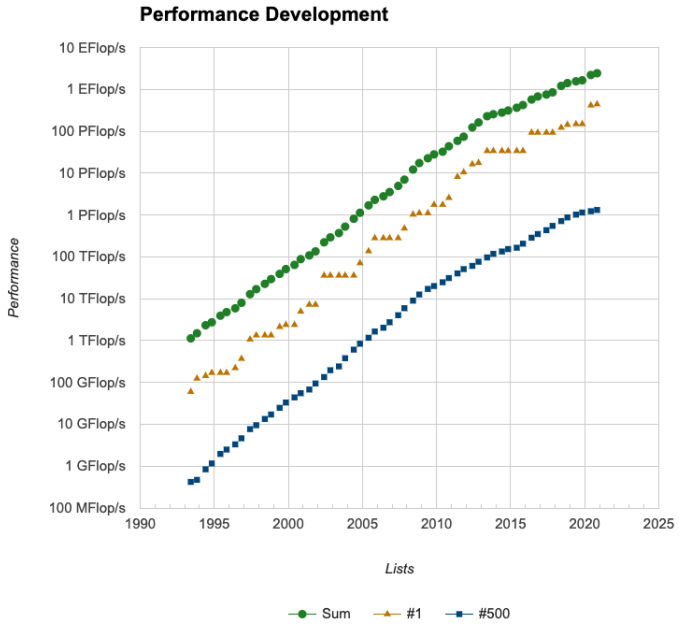

Figura 5: Desarrollo del rendimiento [TOP500].

responsable durante toda la vida útil del sistema. En el corazón del sistema HPC se encuentra la estructura y organización de sus innumerables componentes y la semántica o reglas mediante las cuales operan y realizan las aplicaciones de usuario que se les ofrecen. Incluso más que el hardware, el sistema HPC es una amplia gama de componentes de software que controlan la jerarquía de los componentes físicos y administran las cargas de trabajo del usuario.

En un sentido importante, el sistema informático de alto rendimiento tiene una funcionalidad básica y subsistemas en común con una computadora personal portátil. Estas capacidades principales, compartidas por ambos extremos, incluyen las siguientes:

- las funciones operativas que transforman los valores de los datos de entrada en resultados de salida
- la memoria interna que almacena los datos sobre los que opera el sistema
- los canales de comunicación a través de los cuales se transfieren datos intermedios entre diferentes componentes y subsistemas durante la ejecución de la aplicación
- el hardware de control que coordina la interoperabilidad entre los componentes constituyentes y subsistemas
- el almacenamiento masivo que organiza y mantiene los datos persistentes, el software del sistema y los programas

- los canales e interfaces de entrada/salida que conectan a los usuarios con el sistema

De manera similar, el software de un sistema HPC tiene mucho en común con un servidor empresarial. La supercomputadora tiene una estructura de software que sirve para muchos de los mismos propósitos de interfaz, control y funcionalidad, incluidos, entre otros, los siguientes:

- el sistema operativo que gestiona todos los aspectos de la máquina y su funcionamiento
- los compiladores que traducen programas de aplicación escritos en lenguajes sintácticos legibles por humanos (y otras interfaces) a código binario legible por máquina
- sistemas de archivos que presentan una abstracción lógica de almacenamiento masivo y organizan los datos en dispositivos de almacenamiento
- la inumerable cantidad de controladores de software de los dispositivos de E/S mediante los cuales la computadora se comunica con el mundo externo y usuarios
- las muchas herramientas que componen gran parte de los entornos de usuario esperados

Lo que distingue a un sistema HPC de una computadora convencional es la organización, la interconectividad y la escala de los recursos de los componentes y la capacidad del software de soporte para administrar la operación del sistema a esa escala. Por escala se entiende el grado de paralelismo físico y lógico, es decir, la replicación de componentes físicos claves como procesadores y bancos de memoria y la delimitación de una serie de tareas a realizar simultáneamente. Si bien incluso una computadora portátil de un solo *socket* incorpora cierto paralelismo, un sistema HPC está estructurado en muchos más niveles, cada uno de los cuales suele ser mucho más sustancial. Es esta organización paralela, los métodos mediante los cuales los subsistemas constituyentes se coordinan para resolver un problema compartido, y la funcionalidad adicional del software del sistema y los modelos de programación que proporcionan dicha gestión, lo que diferencia a la supercomputadora de sus contrapartes más pequeñas. Desde el punto de vista del programador, es la necesidad de pensar en paralelo (muchas cosas suceden al mismo tiempo) y distribuidas (cosas que suceden en diferentes lugares separados por la distancia) lo que diferencia a la supercomputadora de la computadora del día a día. Esto requiere conocimiento y habilidad en el empleo de interfaces de programación que exponen y explotan el paralelismo de las aplicaciones y los algoritmos que permiten la operación simultánea de muchas partes del cálculo que contribuyen a la respuesta final.

2.2.5 *Problemas de supercomputación*

El campo de la supercomputación nació en medio de los avances en la investigación nuclear experimental, y desde entonces ha crecido hasta afectar casi todos los campos de investigación impulsados por el experimento. Debido a que la génesis de la supercomputación radica en la simulación de problemas impulsados por la física nuclear, muchos problemas de supercomputación se enmarcan en el contexto del seguimiento de grandes sistemas de partículas, que consisten en diferentes especies que pueden interactuar entre sí y no están en equilibrio. Tales problemas de no equilibrio son generalmente difíciles de calcular analíticamente y pueden ser muy costosos de explorar experimentalmente. En consecuencia, este tipo de problemas aparecen con frecuencia en las supercomputadoras, debido a la capacidad de sondeo de alta resolución de la simulación y al costo sustancialmente reducido al que se puede realizar el experimento computacional.

Otra clase de problema de supercomputación, que se superpone con el seguimiento de grandes sistemas de partículas, son aquellos que resuelven un conjunto de ecuaciones diferenciales parciales. Por ejemplo, una gran fracción del tiempo de supercomputación se dedica a resolver las ecuaciones de Navier-Stokes para el flujo de fluído, debido a su relevancia para muchos problemas de ingeniería. Como segundo ejemplo, la detección directa de una fuente astrofísica de radiación gravitacional en 2015 por parte de *LIGO Scientific Collaboration* [19], fue respaldada por millones de horas de recursos de supercomputación que resolvieron las ecuaciones de campo de Einstein para simular la fusión de agujeros negros binarios.

Muchas clases de problemas de HPC están diseñadas en torno a la capacidad de la supercomputadora para resolver problemas en álgebra lineal. En ciencia e ingeniería, el resultado de discretizar ecuaciones diferenciales parciales, con frecuencia resulta en un sistema de ecuaciones lineales. Esto ha llevado al desarrollo de técnicas de solución directa e iterativa para supercomputadoras. El ya mencionado LINKPACK, es un problema de álgebra lineal que trabaja con matrices densas.

Si bien muchos problemas de HPC surgen de modelos matemáticos, algunos de los problemas de supercomputación más importantes hoy en día surgen de problemas de gráficos. Los problemas gráficos a menudo provienen de problemas que surgen en la gestión del conocimiento, la inteligencia artificial, la lingüística, las redes, la biología y los sistemas dinámicos. La variedad y la novedad de los problemas de supercomputación continúan expandiéndose mucho más allá de sus raíces de física nuclear. A medida que los conjuntos de habilidades y recursos de supercomputación se vuelven cada vez más comunes, es difícil imaginar un campo analítico que no se vea afectado por HPC en el futuro.

2.2.6 *Programación de aplicaciones*

La visión principal que tiene el usuario de un sistema HPC es a través de una o más interfaces de programación, que toman la forma de lenguajes de programación, bibliotecas u otros servicios. Estos se expanden mediante conjuntos adicionales de herramientas que ayudan a crear, optimizar y depurar códigos de aplicación.

La programación en el régimen de supercomputación tiene requisitos y características adicionales. El requisito principal es el rendimiento, que es lo que diferencia a la programación de HPC de otros dominios. Este requisito sólo es superado por la corrección y la repetibilidad. La *performance* está representada por la necesidad de explotación del paralelismo computacional. El procesamiento paralelo implica, la definición de tareas paralelas, estableciendo los criterios que determinan cuándo se realiza una tarea, la sincronización entre tareas para coordinar el intercambio y la asignación a recursos informáticos.

Un segundo aspecto importante de la programación para HPC, es el control de las asignaciones de datos y tareas, a los recursos físicos de los sistemas paralelos y distribuidos. La naturaleza del paralelismo puede variar significativamente, dependiendo de la forma de la arquitectura del sistema informático que se utilizará.

Por otro lado, son importantes el determinismo, la corrección, la depuración y la portabilidad. Dependiendo de la naturaleza de la arquitectura, se emplean diferentes modelos de programación. Una dimensión de diferenciación es la granularidad del flujo de trabajo paralelo. El paralelismo de grano fino enfatiza en las interfaces de programación de sistemas de memoria compartida de múltiples hilos, como OpenMP. El paralelismo de grano medio a grueso, como lo reflejan los procesadores y clústeres paralelos masivos (MPP) altamente escalados, se representa principalmente mediante procesos secuenciales de comunicación, como la interfaz de paso de mensajes (MPI) y sus muchas variantes.

3 MODELOS COMPUTACIONALES: CPU VS GPU

En este capítulo se explicarán los conceptos básicos sobre las arquitecturas de la CPU y la GPU, relevantes para la computación de alto desempeño y particularmente para este trabajo. Además, se hará hincapié en los detalles más importantes del diseño y la implementación de CUDA, la plataforma de desarrollo para GPUs de NVIDIA.

3.1 CPU

Una unidad central de procesamiento (CPU), es el circuito electrónico, dentro de una computadora, que ejecuta instrucciones que conforman un programa. La CPU realiza operaciones básicas de aritmética, lógica, control y entrada/salida especificadas por las instrucciones del programa. Los componentes principales de una CPU incluyen la unidad de lógica aritmética (ALU) que, como su nombre lo indica, realiza operaciones aritméticas y lógicas; los registros del procesador, que suministran operandos a la ALU y almacenan los resultados de las operaciones de ALU, y una unidad de control que orquesta la recuperación (desde la memoria) y la ejecución de instrucciones dirigiendo las operaciones coordinadas de la ALU, registros y otros componentes.

3.1.1 *Jerarquía de memoria*

La figura 6 traza las proyecciones de rendimiento de un solo procesador, contra la mejora histórica del rendimiento para acceder a la memoria principal. La línea del procesador muestra el aumento de las solicitudes de memoria por segundo, en promedio, mientras que la línea de memoria muestra el aumento de los accesos DRAM por segundo, suponiendo una sola DRAM y un solo banco de memoria. Es debido a esta brecha existente entre los rendimientos del procesador vs el acceso a memoria principal, lo que hace necesaria una jerarquía de memoria.

Los pioneros informáticos predijeron correctamente que los programadores querrían cantidades ilimitadas de memoria rápida. Una solución económica para ese deseo es una jerarquía de memoria que aproveche la localidad y las compensaciones en el costo-rendimiento de las tecnologías de memoria. El **principio de localidad** es la tendencia de un procesador a acceder al mismo conjunto de ubicaciones de memoria de forma repetitiva durante un corto período de tiempo. Existen dos tipos básicos de localidad de referencia: localidad temporal y espacial. La localidad temporal se refiere a la reutilización de datos y/o recursos específicos dentro de un período de tiempo relativamente pequeño. La localidad espacial (también denominada localidad de datos) se refiere al uso de elementos de datos dentro de

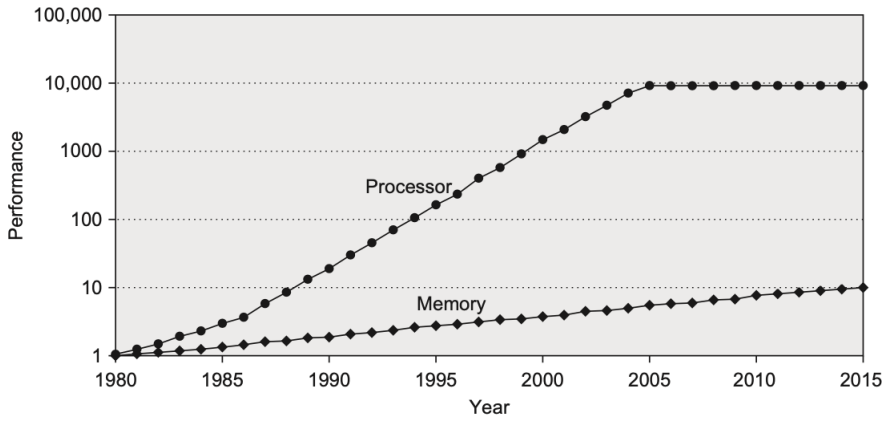

Figura 6: Rendimiento de un CPU vs. memoria RAM [21].

ubicaciones de almacenamiento relativamente cercanas. Este principio, más la directriz de que para una tecnología de implementación y un presupuesto de energía dados, un hardware más pequeño puede hacerse más rápido, condujo a jerarquías basadas en memorias de diferentes velocidades y tamaños. La figura 7 muestra varias jerarquías de memoria multinivel diferentes, incluidos tamaños y velocidades de acceso típicos.

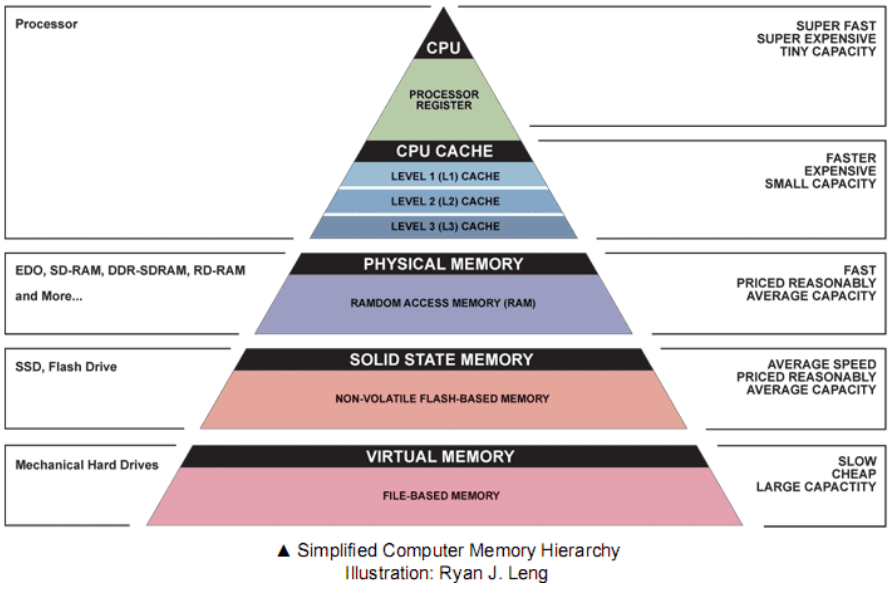

Figura 7: Jerarquía de memoria [31].

Debido a que la memoria rápida es más costosa, una jerarquía de memoria está organizada en varios niveles, cada uno más pequeño, más rápido y más costoso por *byte* que el siguiente nivel inferior, que está más alejado del procesador. El objetivo, es proporcionar un sistema de memoria con un costo por *byte*, casi tan bajo como el nivel de memoria más barato y con una velocidad casi tan rápida como el nivel más rápido.

3.1.1.1 *Registros*

Los registros del procesador están en la parte superior de la jerarquía de memoria y proporcionan la forma más rápida de acceder a los datos. Son una memoria pequeña y rápida que se encuentra integrada dentro de la pastilla del procesador y permite guardar los datos más utilizados. Normalmente se miden por la cantidad de bits que pueden contener, por ejemplo, un "registro de 32 bits".

3.1.1.2 *Caché*

Un caché es una memoria pequeña y rápida, que almacena copias de los datos de las ubicaciones de memoria principal utilizadas con frecuencia, con el fin de reducir los accesos a la misma. Es una memoria de tipo volátil SRAM y se ubica entre la CPU y la memoria principal en la jerarquía de memoria. En SRAM, un bit de datos se almacena usando el estado de una celda de memoria de seis transistores. Esta forma de RAM es más costosa de producir, pero generalmente es más rápida y requiere menos potencia dinámica que la DRAM. Hoy, gracias a la Ley de Moore, todos los niveles de cachés están integrados en el chip del procesador.

Cuando el procesador necesita leer o escribir una ubicación en la memoria, primero verifica si la entrada correspondiente se encuentra en la memoria caché. Si el procesador encuentra que la ubicación de la memoria está en la memoria caché, el valor se lee de ahí mismo produciendose lo que se denomina un *cache hit*. Por el contrario, si el procesador no encuentra la ubicación de la memoria en la caché, se ha producido un *cache miss*. Para este último caso, el procesador escribe inmediatamente los datos en la caché.

Los datos se transfieren entre la memoria y el caché en bloques de tamaño fijo, llamados **líneas de caché** o **bloques de caché**. Cuando una línea de caché se copia de la memoria principal al caché, se crea una entrada de caché. La entrada de caché incluirá los datos copiados, así como la ubicación de memoria solicitada, llamada **etiqueta**. Dentro del bloque se copiarán palabras que no son necesarias en el momento pero que es probable que se necesiten pronto debido a la localidad espacial.

El caché se divide en tres niveles principales, L1, L2 y L3. La jerarquía aquí es de nuevo según la velocidad y, por lo tanto, el tamaño de la memoria caché.

1. **Caché L1**: es la memoria más rápida. En cuanto al tamaño, generalmente ronda los 100 KB y hay una por *core*. Se divide generalmente de dos maneras, en el caché de instrucciones y el caché de datos. El caché de instrucciones trata con la información sobre la operación que la CPU debe realizar, mientras que el caché de datos contiene los datos en los que se realizará la operación.

2. **Caché L2**: es más lento que el caché L1, pero de mayor tamaño. Su tamaño generalmente varía entre 256 KB a 8 MB y hay una

por procesador o compartidas entre estructuras más pequeñas, por ejemplo, pares de cores.

3. **Caché L3**: es la unidad de memoria caché más grande pero también la más lenta. Puede oscilar entre 4 MB y más de 50 MB y hay una por core o una por *chiplet*.

3.1.1.3 *Memoria principal*

La memoria de acceso aleatorio (RAM) es una forma de memoria de computadora que puede leerse y cambiarse en cualquier orden, típicamente utilizada para almacenar datos y programas. Un dispositivo de memoria de acceso aleatorio permite leer o escribir elementos de datos en casi la misma cantidad de tiempo, independientemente de la ubicación física de los datos. Contiene circuitos de multiplexación y demultiplexación, para conectar las líneas de datos al almacenamiento direccionado, para leer o escribir la entrada. Normalmente está asociada con tipos de memoria volátiles, donde la información almacenada se pierde si se corta la alimentación.

Generalmente se utilizan memorias de acceso aleatorio volátil DRAM (memoria dinámica de acceso aleatorio). DRAM almacena un bit de datos utilizando un par de transistores y condensadores que juntos comprenden una celda DRAM. El capacitor tiene una carga alta o baja, y el transistor actúa como un interruptor que permite que los circuitos de control en el chip lean el estado de carga del capacitor o lo cambien. Como esta forma de memoria es menos costosa de producir que la RAM estática, es la forma predominante de memoria de computadora utilizada en las computadoras modernas.

En general, el término RAM se refiere únicamente a dispositivos de memoria de estado sólido (DRAM o SRAM), y más específicamente a la memoria principal en la mayoría de las computadoras.

RAM de datos de gráficos

Las GDRAM (DRAM de gráficos) son una clase especial de DRAM basadas en diseños SDRAM pero diseñadas para manejar las demandas de mayor ancho de banda de las unidades de procesamiento de gráficos. GDDR5 se basa en DDR3 con GDDR anteriores basados en DDR2. Debido a que las unidades de procesador de gráficos requieren más ancho de banda por chip DRAM que las CPU, las GDDR tienen algunas diferencias importantes:

1. Los GDDR tienen interfaces más amplias: 32 bits frente a 4, 8 o 16 en los diseños actuales.

2. Los GDDR tienen una frecuencia de reloj máxima más alta en los pines de datos. Para permitir una mayor tasa de transferencia sin incurrir en problemas de señalización, las GDRAMS normalmente se conectan directamente a la GPU y se conectan soldando a la placa, a diferencia de las DRAM, que normalmente están dispuestas en una matriz expandible de DIMM.

En conjunto, estas características permiten que las GDDR se ejecuten de dos o cinco veces el ancho de banda por DRAM en comparación con las DRAM DDR3.

3.1.1.4 *Escala de latencias*

Si bien el tiempo se puede comparar numéricamente, también ayuda tener un instinto sobre el mismo y las expectativas de latencia de diferentes fuentes. Los componentes del sistema operan en escalas de tiempo muy diferentes en órdenes de magnitud, por lo que puede ser difícil comprender cuán grandes son esas diferencias. En la figura 8, se proporcionan latencias de ejemplo, para demostrar las diferencias en las escalas de tiempo con las que estamos trabajando. La tabla muestra el tiempo promedio que puede tomar cada operación en la vida real.

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

Figura 8: Escala de tiempo de latencias del sistema [20].

En resumen, es importante tener en cuenta todos los detalles discutidos en este capítulo a la hora de programar, dado que el uso de patrones de acceso a la memoria que exploten los niveles de la jerarquía obtendrán una mayor *performance*.

3.2 GPU Y CUDA

CUDA (*Compute Unified Device Architecture*) es una plataforma de propósito general de computación paralela y modelo de programación, desarrollado por NVIDIA. Esta plataforma aprovecha el motor de cálculo paralelo de sus GPUs, para resolver problemas computacionales complejos de una manera más eficiente que una CPU.

CUDA viene acompañada de un entorno de software que permite a los desarrolladores utilizar C/C++ como lenguaje de programación de alto nivel. Además, se admiten otros lenguajes, interfaces de programación de aplicaciones o enfoques basados en directivas, como FORTRAN, DirectCompute, OpenACC.

3.2.1 *Modelo de programación escalable*

El desafío es desarrollar un software de aplicación que escale de manera transparente su paralelismo, y así aprovechar el creciente número de núcleos de la GPU, de la misma manera que las aplicaciones de gráficos 3D escalan de manera transparente su paralelismo a muchas GPUs, con un número muy variable de núcleos. El modelo de programación paralela CUDA, está diseñado para superar este desafío, mientras mantiene una curva de aprendizaje baja para programadores familiarizados con lenguajes de programación estándar como C++.

En esencia, existen tres abstracciones claves: una jerarquía de grupos de hilos, memorias compartidas y sincronización de barrera, que simplemente se exponen al programador como un conjunto mínimo de extensiones del lenguaje. Estas extensiones del lenguaje proporcionan paralelismo de datos de grano fino y paralelismo de hilos, anidados dentro de un paralelismo de datos de grano grueso y paralelismo de tareas. Además, guían al programador a dividir el problema en subproblemas gruesos que pueden resolverse independientemente en paralelo, mediante bloques de hilos, y cada subproblema en piezas más finas que pueden resolverse cooperativamente en paralelo, mediante todos los hilos dentro del bloque.

Esta descomposición preserva la expresividad del lenguaje al permitir que los hilos cooperen al resolver cada subproblema y, al mismo tiempo, permite la escalabilidad automática. De hecho, cada bloque de hilos puede programarse, planificarse, ejecutarse en cualquiera de los multiprocesadores disponibles dentro de una GPU, en cualquier orden, de forma simultánea o secuencial, de modo que un programa CUDA compilado pueda ejecutarse en cualquier número de multiprocesadores como se ilustra en la figura 9.

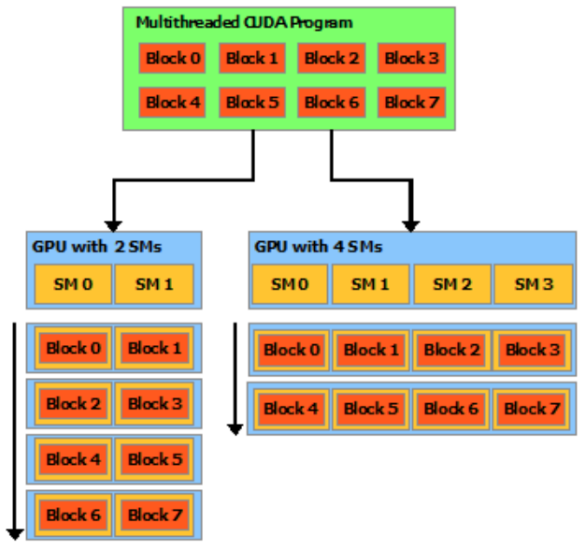

Figura 9: Escalabilidad automática [36].

3.2.1.1 *Kernels*

CUDA C++ extiende de C++, permitiendo que el programador defina funciones de C++ llamadas *kernels*, que se ejecutan N veces en paralelo por N hilos de CUDA diferentes, en lugar de solo una vez como las funciones normales de C++. Además, el programador debe definir el valor N para la ejecución del *kernel*. Cada hilo que ejecuta el *kernel* recibe un ID (identificador único) que es accesible por cada hilo dentro del *kernel* a través de variables integradas.

3.2.1.2 *Jerarquía de Hilos*

Por conveniencia, cada identificador de hilos es un vector de 3 componentes, de modo que los hilos pueden identificarse utilizando un índice de hilo unidimensional, bidimensional o tridimensional, formando un bloque de hilos. Esto proporciona una forma natural de invocar el cálculo a través de los elementos en un dominio, como un vector, matriz o volumen respectivamente. Existe un límite en el número de hilos por bloque, ya que se espera que todos los hilos de un bloque residan en el mismo núcleo del procesador y deben compartir los recursos de memoria limitados de ese núcleo. En las GPU actuales, un bloque puede contener hasta 1024 hilos. Los bloques se organizan en una grilla de una, dos o tres dimensiones de bloques de hilos como se ilustra en la figura 10.

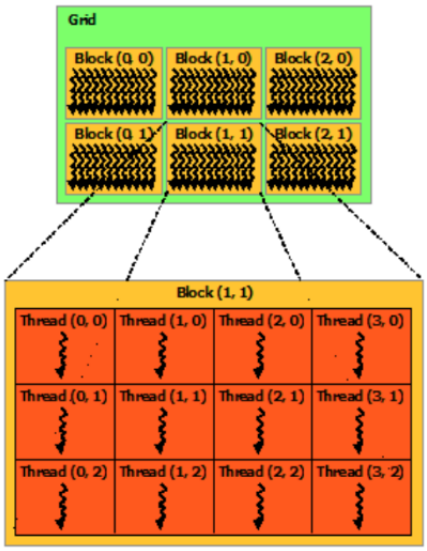

Figura 10: Grilla de bloques de hilos [36].

El número de bloques de hilos en una grilla generalmente está dictado por el tamaño de los datos que se procesan.

Los hilos dentro de un bloque pueden cooperar compartiendo datos a través de alguna memoria compartida, y sincronizando su ejecución para coordinar los accesos a la memoria. Más precisamente, se puede especificar puntos de sincronización en el *kernel*, que actúan como una barrera en la que todos los hilos del bloque deben esperar antes de poder continuar. Para una cooperación eficiente, se espera que la

memoria compartida sea una memoria de baja latencia, cerca de cada núcleo de procesador y que la sincronización de hilos no tenga mucha penalización.

3.2.1.3 *Jerarquía de Memoria*

Los hilos de CUDA pueden acceder a datos de múltiples espacios de memoria durante su ejecución, como se ilustra en la figura 11.

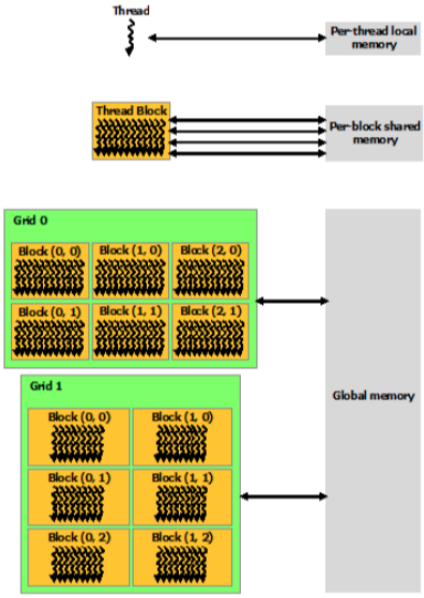

Figura 11: Jerarquía de memoria [36].

Cada hilo tiene memoria local privada. Cada bloque de hilos tiene memoria compartida, visible para todos los hilos del bloque y con la misma vida útil que el bloque. Todos los hilos tienen acceso a la misma memoria global. También hay dos espacios de memoria adicionales de solo lectura, accesibles por todos los hilos: los espacio de memoria constante y de textura. Los espacios de memoria global, constante y de textura están optimizados para diferentes usos. La memoria de textura también ofrece diferentes modos de direccionamiento, así como el filtrado de datos, para algunos formatos de datos específicos. Los espacios de memoria global, constante y de textura son persistentes en los lanzamientos de kernel por la misma aplicación.

3.2.1.4 *Programación Heterogénea*

Como se ilustra en la figura 12, el modelo de programación CUDA supone que los hilos se ejecutan en un dispositivo físicamente separado, que funciona como co-procesador para el *host*, que ejecuta el programa C++. Este proceso de realiza de manera asincrónica entre el *host* y el *device*. Por ejemplo, en el caso de este trabajo, cuando los *kernels* se ejecutan en una GPU, y el resto del programa C++ se ejecuta en una CPU. Además, este modelo supone también que tanto el *host* como el dispositivo, mantienen sus propios espacios de memoria separados en DRAM, por lo tanto, un programa gestiona los espacios

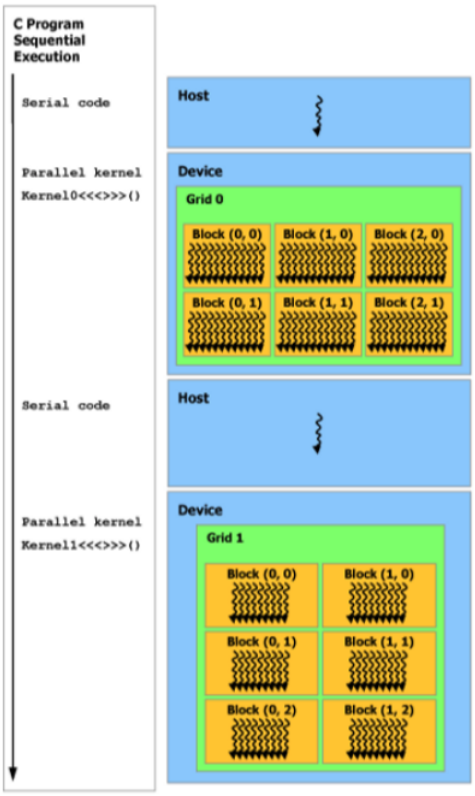

Figura 12: Programación heterogénea [36].

de memoria global, constante y de textura, visibles para los *kernels* a través de llamadas a la API de CUDA en *runtime*. Esto incluye la asignación y desasignación de memoria del dispositivo, así como la transferencia de datos entre el *host* y la memoria del dispositivo.

3.2.2 *Implementación de Hardware*

La arquitectura de la GPU NVIDIA, se basa en una matriz escalable de transmisión multiproceso llamada *Streaming Multiprocessors* (SM). Un multiprocesador está diseñado para ejecutar cientos de hilos simultáneamente. Para administrar una cantidad tan grande de hilos, emplea una arquitectura única llamada SIMT (*Single-Instruction, Multiple-Thread*). Las instrucciones están canalizadas, aprovechando el paralelismo a nivel de instrucción dentro de un solo hilo, así como un extenso paralelismo a nivel de hilo.

3.2.2.1 *Aquitectura SIMT*

El multiprocesador crea, gestiona, programa y ejecuta hilos en grupos llamados *warps*. Un *warp* es una abstracción de CUDA sobre las unidades de vectores. Actualmente, la cantidad de hilos paralelos de un *warp* es 32, pero esto no siempre fue así. Los hilos individuales que componen un *warp* comienzan juntos en la misma dirección del programa, pero tienen su propio contador de direcciones de instrucción y

estado de registro y, por lo tanto, son libres de ramificarse y ejecutarse independientemente.

Cuando un multiprocesador recibe uno o más bloques de hilos para ejecutar, los divide en *warps* y cada *warp* es programado por un planificador para su ejecución. La forma en que un bloque se divide en *warps* es siempre la misma; cada *warp* contiene hilos de identificaciones numéricos consecutivos, el primer *warp* con identificadores de 0 a 31, el segundo de 32 a 63, etc.

Un *warp* ejecuta una instrucción común a la vez, por lo que se obtiene una eficiencia total cuando los 32 hilos de un *warp* coinciden en su ruta de ejecución. Si los hilos de un *warp* divergen a través de una rama condicional, el *warp* ejecuta cada ruta de rama tomada, deshabilitando los hilos que no están en esa ruta. Al converger, todos los hilos vuelven al mismo punto del programa, esto es, para hilos de un mismo *warp*. Diferentes *warps* ejecutan independientemente.

Para propósitos de corrección, el programador puede ignorar esencialmente el comportamiento SIMT; sin embargo, se pueden lograr mejoras sustanciales en el rendimiento, teniendo cuidado de que el código rara vez requiera hilos en una distorsión para diverger. En la práctica, esto es análogo al papel de las líneas de caché en el código tradicional: el tamaño de la línea de caché se puede ignorar de forma segura cuando se diseña para la corrección, pero debe considerarse en la estructura del código cuando se diseña para obtener el máximo rendimiento.

3.2.2.2 *Scheduling*

Cuando un programa CUDA en la CPU del *host* invoca una grilla del *kernel*, los bloques de la grilla se enumeran y distribuyen a multiprocesadores con capacidad de ejecución disponible. Los hilos de un bloque se ejecutan simultáneamente en un multiprocesador, y varios bloques de hilos pueden ejecutarse simultáneamente en un multiprocesador. A medida que los bloques terminan, se lanzan nuevos en los multiprocesadores desocupados.

3.2.2.3 *Hardware Multihilo*

El contexto de ejecución (contadores de programas, registros, etc.) para cada *warp*, procesado por un multiprocesador, se mantiene en el chip durante toda la vida útil del *warp*. Por lo tanto, cambiar de un contexto de ejecución a otro no tiene costo, y en cada momento de emisión de instrucciones, un planificador de *warp* selecciona un *warp* que tiene hilos listos para ejecutar su próxima instrucción (los hilos activos del *warp*), y emite la instrucción a esos hilos.

En particular, cada multiprocesador tiene un conjunto de registros de 32 bits que se dividen entre los *warps*, y una caché de datos paralelo o memoria compartida, que se divide entre los bloques de hilos. El número de bloques y *warps* que pueden residir y procesarse juntos en el multiprocesador, para un núcleo determinado, depende de la cantidad de registros y memoria compartida utilizada por el *kernel*,

además de la cantidad de registros y memoria compartida disponibles en el multiprocesador. Por otro lado, existe un número máximo de bloques y *warps* residentes por multiprocesador. Estos límites, así como la cantidad de registros y la memoria compartida disponibles en el multiprocesador, son una función de la capacidad de cálculo del dispositivo. Si no hay suficientes registros o memoria compartida disponibles por multiprocesador para procesar al menos un bloque, el *kernel* no se iniciará.

La fórmula para calcular el número total de *warps* en un bloque es $\mathtt{ceil}(T \times W_{\mathtt{size}}, 1)$, donde:

- T es el número de hilos por bloque
- $W_{\mathtt{size}}$ es el tamaño de *warp*, que es igual a 32
- $\mathtt{ceil}(x, y)$ es igual a x redondeado al múltiplo más cercano de y

3.2.3 *Técnicas de performance*

Para conseguir el máximo rendimiento posible de la GPU, es necesario seguir ciertas técnicas en el uso de la aquitectura. La optimización del rendimiento gira en torno a tres estrategias básicas:

- Maximizar la ejecución paralela para lograr la máxima utilización
- Optimizar el uso de memoria para lograr el máximo rendimiento de memoria
- Optimizar el uso de la instrucción para lograr el máximo rendimiento

Las estrategias que producirán la mejor ganancia de rendimiento, para una parte particular de una aplicación, dependen de los limitadores de rendimiento para esa parte; optimizar el uso de instrucciones de un *kernel*, que está limitado principalmente por los accesos a la memoria, no producirá ninguna ganancia de rendimiento significativa, por ejemplo.

Para **maximizar la utilización**, la aplicación debe estructurarse de manera que exponga el mayor paralelismo posible, y mapee este paralelismo de manera eficiente a los diversos componentes del sistema, para mantenerlos ocupados la mayor parte del tiempo, haciendo un uso efeiciente de los recursos compartidos de cada *Streaming Multiprocessor* como registros, *shared memory*, etc. La aplicación debe maximizar la ejecución paralela entre el *host*, los dispositivos y el *bus* que conecta el *host* a los dispositivos, utilizando llamadas y flujos de funciones asíncronas. Debe asignar a cada procesador el tipo de trabajo que mejor realiza: cargas de trabajo en serie al *host*; cargas de trabajo paralelas a los dispositivos. Se deben minimizar las bifurcaciones y las sincronizaciones como barreras o *mutex* de escritura de memoria.

El primer paso para **maximizar el rendimiento general de la memoria** para la aplicación, es minimizar las transferencias de datos con

poco ancho de banda. Eso significa minimizar las transferencias de datos entre el *host* y el dispositivo, ya que tienen un ancho de banda mucho menor que las transferencias de datos entre la memoria global y el dispositivo.

La memoria compartida es equivalente a un caché administrado por el usuario: la aplicación lo asigna y accede explícitamente a él. Un patrón de programación típico, es organizar los datos que provienen de la memoria del dispositivo en la memoria compartida. El esquema es el siguiente:

1. Cargar datos desde la memoria del dispositivo a la memoria compartida
2. Sincronizar con todos los otros subprocesos del bloque, para que cada subproceso pueda leer de forma segura las ubicaciones de memoria compartida que fueron pobladas por diferentes subprocesos
3. Procesar los datos en la memoria compartida
4. Sincronizar nuevamente si es necesario, para asegurarse de que la memoria compartida se haya actualizado con los resultados
5. Escribir los resultados en la memoria del dispositivo

El siguiente paso para maximizar el rendimiento de la memoria, es organizar los accesos en función de los patrones óptimos.

La memoria global reside en la memoria del dispositivo, el acceso a la memoria del dispositivo se realiza a través de transacciones de 32, 64 o 128 bytes. Cuando un *warp* ejecuta una instrucción que accede a la memoria global, fusiona los accesos de memoria de los hilos dentro del *warp*, en una o más de estas transacciones de memoria, dependiendo del tamaño de la palabra a la que accede cada hilo y de la distribución de las direcciones de memoria en hilos. En general, cuantas más transacciones sean necesarias, más palabras no utilizadas se transfieren, lo que reduce el rendimiento de la instrucción en consecuencia.

Los accesos a la memoria local solo ocurren para algunas variables automáticas. El espacio de memoria local, reside en la memoria del dispositivo, por lo que los accesos a la memoria local tienen la misma latencia alta y ancho de banda bajo que los accesos a la memoria global, y están sujetos a los mismos requisitos para la fusión de la memoria. Sin embargo, la memoria local está organizada de manera tal, que los *warps* de hilos consecutivos acceden a palabras consecutivas de 32 bits. Por lo tanto, los accesos están completamente unidos, siempre que todos los hilos de un *warp* accedan a la misma dirección relativa.

La memoria compartida tiene un ancho de banda mucho mayor, y una latencia mucho menor que la memoria local o global, debido a que está en el chip. Para obtener el máximo rendimiento, es importante comprender cómo las direcciones de memoria se asignan a los bancos de memoria para programar las solicitudes de memoria, a fin de minimizar los conflictos bancarios.

El espacio de memoria constante reside en la memoria del dispositivo, y se almacena en caché en la memoria caché constante. Los espacios de memoria de textura y superficie residen en la memoria del dispositivo, y se almacenan en la memoria caché de textura. Leer la memoria del dispositivo a través de la obtención de texturas o superficies, presenta algunos beneficios que pueden convertirlo en una alternativa ventajosa para leer la memoria del dispositivo desde la memoria global o constante.

Para **maximizar el rendimiento de la instrucción**, la aplicación debe:

- Minimizar el uso de instrucciones aritméticas con bajo rendimiento,
- Minimizar los *warps* divergentes causados por las instrucciones de control de flujo, y
- Reducir la cantidad de instrucciones

3.2.4 *Pascal vs. Turing*

Cabe mencionar, que al desarrollar esta versión se encontraron resultados distintos corriendo en **zx81** (*arquitectura Pascal*) vs **jupiterace** (*arquitectura Turing*). Esto se debe a las diferencias existentes entre las dos arquitecturas NVIDIA, las cuales se explicarán a continuación.

3.2.4.1 *Arquitectura Turing*

El *Streaming Multiprocessor* Turing está dividido en cuatro bloques de procesamiento, cada uno con 16 FP32 *cores*, 16 INT32 *cores*, dos tensor *cores*, un *warp schedule* y una unidad de *dispatch*. Cada bloque incluye un nuevo caché de instrucciones L0 y un archivo de registro de 64 KB. Los cuatro bloques de procesamiento, comparten un total de 96 KB de caché de datos L1/*shared memory*. Las cargas de trabajo informáticas pueden dividir los 96 KB en 32 KB de *shared memory* y 64 KB caché L1 o *shared memory* de 64 KB y caché L1 de 32 KB.

Tal como se explica en [9], el SM de Turing también presenta una nueva arquitectura unificada para *shared memory*, L1 y textura. Este diseño unificado permite que la caché L1 aproveche los recursos, aumentando su impacto en ancho de banda en 2x por clúster de procesamiento de texturas (TPC), en comparación con **Pascal**, y permite reconfigurarlo para crecer más, cuando las asignaciones de *shared memory* no están utilizando toda la capacidad de memoria compartida. La caché L1 en **Turing** puede tener un tamaño de hasta 64 KB, combinado con una asignación de *shared memory* de 32 KB por SM, o puede reducir a 32 KB, lo que permite utilizar 64 KB de asignación para *shared memory*. La caché L2 de Turing también ha aumentado la capacidad.

La figura 13 muestra cómo la nueva caché de datos L1 combinada y el subsistema de memoria compartida del SM Turing mejora significativamente el rendimiento, al mismo tiempo que simplifica la

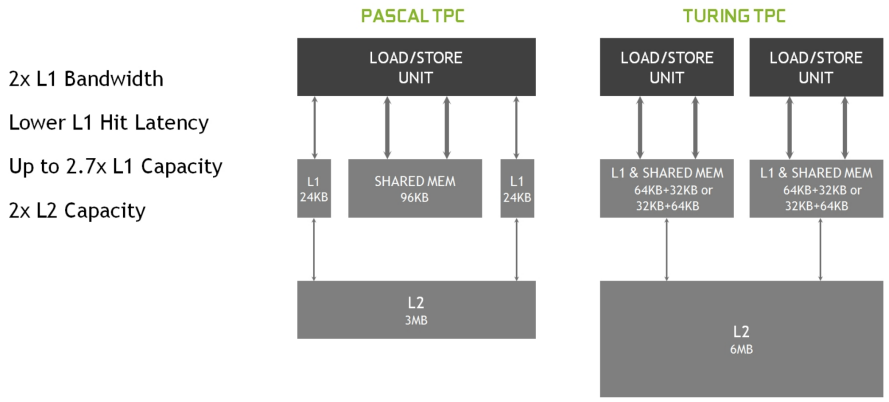

Figura 13: Arquitectura de la memoria compartida en Turing.

programación y reduce el ajuste necesario para alcanzar el rendimiento máximo de la aplicación. Combinando los datos de caché L1 con la memoria compartida, reduce la latencia y proporciona mayor ancho de banda que la implementación de caché L1 utilizada previamente en las GPU Pascal.

3.3 GPU/GPGPU

Se denomina informática de uso general en unidades de procesamiento de gráficos, GPGPU por su siglas en inglés, al uso de GPUs, generalmente utilizadas solo para gráficos de computadora, para realizar computación en aplicaciones tradicionalmente manejadas por la CPU.

Esencialmente, un *pipeline* GPGPU es un tipo de procesamiento paralelo entre una o más GPUs y CPUs, que analizan los datos como si fueran una imagen u otra forma gráfica (matricialmente). Si bien las GPUs funcionan a frecuencias más bajas que un CPU, generalmente tienen una cantidad mayor de *cores*. Por lo tanto, las GPUs pueden procesar muchas más imágenes y datos gráficos por segundo que una CPU tradicional. Migrar datos en forma gráfica y luego usar la GPU para escanear y analizarlos puede crear una gran aceleración.

Estos *pipelines* se desarrollaron a principios del siglo XXI para el procesamiento de gráficos pero se descubrió que satisfacen bien las necesidades de computación científica, y desde entonces se han desarrollado en esta dirección.

3.3.1 *Beneficios de utilizar GPU*

La GPU proporciona un rendimiento de instrucción y un ancho de banda de memoria mucho más alto que la CPU, dentro de un precio y una potencia similares. Muchas aplicaciones aprovechan estas capacidades superiores para ejecutarse más rápido en la GPU que en la CPU.

Esta diferencia en las capacidades entre GPU y CPU existe porque fueron diseñadas con diferentes objetivos. La CPU optimiza para

lograr la menor latencia en cada hilo, extrayendo paralelismo interno para ejecutar fuera de orden, y utilizando caches para ocultar latencia, mientras que la GPU optimiza para lograr el mayor *throughput* posible, utilizando *multithreading* para ocultar la latencia a memoria.

La GPU está especializada en cálculos altamente paralelos y sobretodo homogéneos, que permiten utilizar unidades de vectores anchas, por lo tanto, está diseñada de modo que se dediquen más transistores al procesamiento de datos en lugar del almacenamiento en caché de datos y el control de flujo. La figura 14 muestra un ejemplo de distribución de recursos de chip para una CPU versus una GPU.

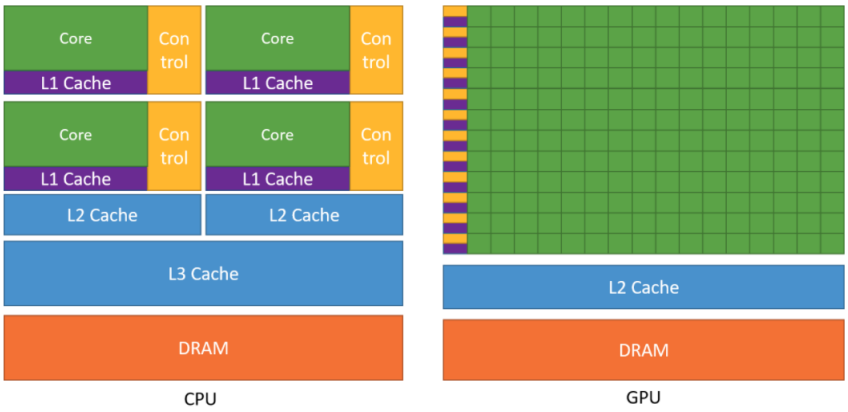

Figura 14: Comparación de las arquitecturas de un CPU vs. GPU [36].

Dedicar más transistores al procesamiento de datos, por ejemplo, cálculos de punto flotante, es beneficioso para cálculos altamente paralelos; la GPU puede ocultar las latencias de acceso a la memoria con el cálculo, en lugar de depender de grandes cachés de datos y control de flujo complejo, para evitar largas latencias de acceso a la memoria, las cuales son costosas en términos de transistores.

En general, una aplicación tiene una combinación de partes paralelas y partes secuenciales, por lo que los sistemas se diseñan como una combinación de GPU y CPU para maximizar el rendimiento general. Las aplicaciones con un alto grado de paralelismo, pueden explotar la naturaleza paralela masiva de la GPU para lograr un mayor rendimiento que en la CPU.

4 OPTIMIZACIONES Y RESULTADOS

En el presente capítulo se explicarán las distintas etapas, llevadas a cabo a lo largo de este trabajo, hasta lograr la optimización definitiva. Junto con las explicaciones se mostrarán los resultados obtenidos en cada etapa.

Para cada optimización, se utilizaron dos simulaciones de distinto tamaño y parámetros de entrada. Estas simulaciones reciben los nombres de ***example 01*** y ***example large***, con 100 y 500 iteraciones respectivamente.

4.1 HARDWARE Y HERRAMIENTAS

A continuación, se describe el *hardware* empleado para la toma de métricas. Se utilizaron los servidores ***jupiterace*** y ***zx81*** de FaMAF, y una macbook pro personal del autor, solo para las versiones de CPU.

Cabe aclarar que, si bien estos servidores fueron los utilizados para la toma de métricas, no fueron los únicos empleados para la etapa de desarrollo. Además de los ya mencionados, se utilizó el servidor *yopi* de FaMAF, *mendieta* de CCAD y unas instancias de *aws* brindadas por Mercadolibre.

Yopi

- CPU:
  - Procesador: Intel(R) Core(TM) i7-7660U CPU 920 @ 2.67GHz
  - Memoria RAM: 16 GB DDR3
  - Arquitectura: Nehalem
- GPU:
  1.
     - Procesador: Tesla K40c
     - Memoria: 12GB GDDR5
     - Arquitectura: Kepler
  2.
     - Procesador: TITAN Xp
     - Memoria: 12GB GDDR5X
     - Arquitectura: Pascal
- Compiladores:
  - gfortran: 9.3.0
  - gcc: 9.3.0
  - nvcc: 11.0.221
- Sistema Operativo: Debian 5.8.7-1 x86_64 GNU/Linux

ZX81

- CPU:
  - Procesador: Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz (14 *cores*, 28 *threads*) x2 unidades
  - Memoria RAM: 128 GB DDR4
  - Arquitectura: Broadwell
- GPU:
  - Procesador: GeForce GTX 1070 x2 unidades
  - Memoria: 8GB GDDR5
  - Arquitectura: Pascal
- Compiladores:
  - gfortran: 9.3.0
  - gcc: 9.3.0
  - nvcc: 11.0.221
- Sistema Operativo: Debian 5.8.7-1 x86_64 GNU/Linux
- *Tools*
  - perf: 5.8.7
  - Intel advisor: 2020 Update 2 (build 606470)

Jupiterace

- CPU:
  - Procesador: Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz (14 *cores*, 28 *threads*) x2 unidades
  - Memoria RAM: 128 GB DDR4
  - Arquitectura: Broadwell
  - *Peak performance*: 2,3 TFLOPS
- GPU:
  - Procesador: GeForce RTX 2080 Ti
  - Memoria: 11GB GDDR6
  - Arquitectura: Turing
  - *Peak performance*: 365,3 GFLOPS
- Compiladores:
  - gfortran: 9.3.0
  - gcc: 9.3.0
  - nvcc: 11.1.74
- Sistema Operativo: Debian 5.7.6-1 x86_64 GNU/Linux
- *Tools*

- perf: 5.7.6
- Intel advisor: 2020 Update 2 (build 606470)
- NVIDIA Nsight Compute: 2020.2.0 (Build 28964561)

MACBOOK

- CPU:
  - Procesador: Intel(R) Core(TM) i7-7660U CPU @ 2.50GHz
  - Memoria RAM: 16 GB 2133 MHz LPDDR3
  - Arquitectura: Kaby Lake
- Compiladores:
  - gfortran: 10.2.0
  - gcc: 10.2.0
- Sistema Operativo: macOS Catalina 10.15.7

Por último, es importante mencionar algunos detalles de los resultados que se verán a continuación. Debido a que la capacidad de CPU en los servidores *jupiterace* y *zx81* es la misma, solo se mostrarán los resultados para uno solo de ellos. Distinto será en los casos de las versiones con GPU, donde el *hardware* cambia mucho, y por ende los resultados también. Por otro lado, dado que la macbook no posee GPU, esta máquina solo será tenida en cuenta para las versiones de CPU.

4.2 *performance* y *roofline*

El modelo de rendimiento de Roofline [52] ofrece una forma intuitiva y perspicaz de extraer características computacionales claves para aplicaciones en computación de alto desempeño. Su capacidad para abstraer la complejidad de las jerarquías de memoria modernas y guiar el análisis del rendimiento y el esfuerzo de optimización ha ganado popularidad en los últimos años.

Roofline es un modelo orientado al rendimiento, centrado en la interacción entre las capacidades computacionales, el ancho de banda de la memoria y la ubicación de los datos. La localidad de los datos hace referencia a la reutilización de los mismos una vez que se cargan desde la memoria principal, y se expresa comúnmente como la **intensidad aritmética** (AI), relación entre las operaciones de punto flotante realizadas y los datos transferidos desde y hacia memoria principal (FLOPs/Byte) para realizar dichas operaciones. El rendimiento (FLOPs/s) está acotado por los dos términos siguientes:

$$\text{GFLOPS/s} \leqslant \min \begin{cases} \text{Peak FLOPS/s} \\ \text{Peak GB/s} \times \text{Arithmetic Intensity} \end{cases}$$

Se pueden identificar los siguientes componentes en un gráfico de *roofline*, los mismos se muestran en la figura 15:

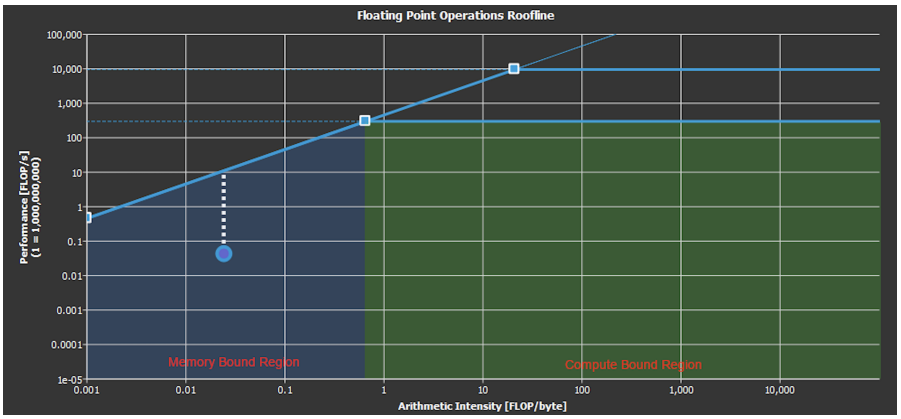

Figura 15: Ejemplo de un gráfico de *roofline* en NVIDIA Nsight Compute [38].

- **Eje vertical**: representa las operaciones de punto flotante por segundo (FLOPS). Este número puede ser bastante grande, es por esto que este eje se representa utilizando una escala logarítmica.
- **Eje horizontal**: representa la intensidad aritmética, que es la relación entre el trabajo (expresado en operaciones de punto flotante) y el tráfico de memoria (expresado en bytes por segundo). La unidad resultante está en operaciones de punto flotante por byte. Este eje también se muestra usando una escala logarítmica.
- **Límite de ancho de banda de memoria**: es la parte inclinada del gráfico. Esta pendiente está determinada por completo por la tasa de transferencia de memoria.
- **Límite de rendimiento máximo**: es la parte plana del gráfico. Este valor está determinado por completo por el rendimiento máximo de la CPU/GPU.
- **Punto de cresta**: es el punto en el que el límite del ancho de banda de la memoria se encuentra con el límite de rendimiento máximo. Este punto es una referencia útil al analizar el rendimiento.
- **Valor logrado**: representa el rendimiento actual.

Como se muestra en la figura 15, el punto de la cresta divide el gráfico de *roofline* en dos regiones. El área sombreada en azul, debajo del límite de ancho de banda de memoria inclinado, es la región de límite de memoria, mientras que el área sombreada en verde, debajo del límite de rendimiento máximo, es la región de límite de cálculo. La región en la que cae el valor alcanzado determina el factor limitante actual del rendimiento.

La distancia desde el valor alcanzado hasta el límite de la línea de techo (que se muestra en esta figura como una línea blanca de puntos) representa la oportunidad de mejorar el rendimiento, manteniendo la intensidad aritmética. Cuanto más cerca esté el valor alcanzado del límite de la línea del techo, más óptimo será el rendimiento. Un

valor alcanzado que se encuentra en el límite del ancho de banda de la memoria, pero que aún no está a la altura del punto de la cresta, indicaría que cualquier mejora adicional en los FLOPS generales solo es posible si se aumenta la intensidad aritmética al mismo tiempo.

Convencionalmente, el modelo *roofline* se centra en un nivel del sistema de memoria, pero esto se ha extendido a toda la jerarquía de memoria en los últimos años, dando origen al *modelo jerárquico de roofline*. *Roofline* jerárquico ayuda a comprender la reutilización de la memoria caché y la ubicación de los datos y proporciona información adicional sobre la eficiencia de la utilización del subsistema de memoria por parte de la aplicación. El *roofline* jerárquico se ha integrado en Intel Advisor [23] [22] y NVIDIA Nsight Compute [37] [38], aunque también es posible calcularlo utilizando herramientas no tan modernas, como es el caso de Intel VTune [24] para CPU y NVIDIA Nvprof [39] para GPU.

En el caso de este trabajo final, se utilizó mucho *roofline* como una herramienta de guía para saber que optimizaciones realizar. Particularmente, se utilizó Intel Advisor y NVIDIA Nsight Compute, ya que las mismas estaban disponibles en *jupiterace*.

4.2.1 *Performance pico*

***Performance* pico**			
hardware	*Memory peak* (GB/s)	FP64 *peak* (TFLOPS)	FP32 *peak* (TFLOPS)
Xeon E5-2680	76,8	0,27	0,54
GeForce RTX 2080 Ti	616	0,42	13,45
GeForce GTX 1070	256,3	0,20	6,46
Tesla K40c	288,4	1,68	5,04

Cuadro 1: *Performance* pico.

4.3 V0: CÓDIGO ORIGINAL

Al código original se le realizaron algunas modificaciones para poder correr en el entorno disponible. Como principal cambio, se utilizaron compiladores GNU, es decir, **gfortran** y **gcc**, para todas las simulaciones. Es por esto, que fue necesario definir una función de tiempo, que reemplace a la utilizada originalmente **DCLOCK**, de la libería *IFPORT* de Intel. Para esto, se utilizó la función *omp_get_wtime()* de la API de OpenMP. Además, se cambió la interfaz de la misma, para que los tiempos se mostraran en segundos en lugar de milisegundos.

Por otro lado, se definieron distintos escenarios de tests para asegurar la correctitud de las optimizaciones realizadas. La *suit* de tests fue escrita en el lenguage *Python*.

Por último, se utilizó la herramienta *cmake* [8] para facilitar, no solo el proceso de compilación, sino que también los de optimización y toma de métricas mediante la definición de comandos definidos por el desarrollador.

Antes de comenzar a optimizar, lo primero que se debe hacer es identificar correctamente las partes de código que vale la pena mejorar, es decir, aquellas que permitirán obtener una ganancia efectiva. Existen múltiples herramientas que facilitan esta búsqueda, algunas de las utilizadas en este trabajo son las ya mencionadas **VTune** [24], **Advisor** [23] de Intel, **Nvprof** [39] y **Nsight Compute** [37] de NVIDIA, y **Perf** [54], una herramienta *Open Source* que forma parte del paquete *linux-common-tools*. Estos *profilers* ayudaron a identificar los cuellos de botella de las simulaciones y guiaron algunas optimizaciones.

Dentro de las funciones más costosas, se encontraban: **vortex_line** (1) y **convect** (2). Donde la primera es llamada mayoritariamente dentro de la segunda. Estas dos funciones sumadas, consumían aproximadamente el 89 % de los recursos computacionales de la simulación en ambos casos, es decir, tanto para el *example o1* como para el *example large*, tal como se puede apreciar en (16).

4.3.1 *Algoritmos: vortex line y convect originales*

Algoritmo 1: *Vortex line.*

Input: R_P, R_1, R_2, G, delta
Result: V

1 $r1 = R\_P - R\_1$
2 $r2 = R\_P - R\_2$
3 $r3 = r1 \times r2$
4 $L = R\_2 - R\_1$
5 $num = (\|r1\| + \|r2\|) * r3$
6 $den = (\|r1\| * \|r2\|) * (\|r1\| * \|r2\| + (r1 \cdot r2)) + (delta * \|L\|)^2$
7 $V = \frac{G}{4*\pi} * \frac{num}{den}$

Donde R_P, R_1, R_2 son vectores de tres dimensiones, G y delta son *doubles*, $\times$ es el *cross product* y $\|x\|$ es la norma del vector x.

Algoritmo 2: *Convect*.

Input: $nnpwkt, nvlls, nvlcv, nvlwkt, GWK, rnpwk, rnpwknew, \ldots$
Result: $GWK, rnpwk, rnpwknew$

```
for i ← 1 to nnpwkt do
    init V
    get R_P
    for j ← 1 to nvlls do
        get R_1, R_2, GLSvl y cutoff
        V ← vortex_line(R_P, R_1, R_2, GLSvl, cutoff)
    if it != 1 then
        for j ← nvlcv + 1 to nvlwkt do
            get R_1, R_2, GWKvl y cutoff
            V ← V+ vortex_line(R_P, R_1, R_2, GWKvl, cutoff)
    V ← V + Vinf
    update rnpwknew
if it != 1 then
    GWK shifter
GWK gather
rnpwknew gather
rnpwknew shifter
rnpwk ← rnpwknew
```

Donde:

- nvlwkt: **N**umber of **V**ortex **L**ines in the **W**a**K**e at **T**ime t, es decir, número de *vortex lines* en la estela en el tiempo t.
- nvlls: **N**umber of **V**ortex **L**ines in the **L**ifting **S**urface, es decir, número total de *vortex lines* en la superficie sustentadora.
- nvlcv: **N**umber of **V**ortex **L**ines to be **C**on**V**ected, es decir, número de *vortex lines* a convectar.
- nnpwkt: **N**umber of **N**odal **P**oints in the **W**a**K**e at **T**ime t, es decir, número de *nodal points* en la estela en el tiempo t.
- GWK: circulaciones del anillo de vórtice de la estela.
- rnpwk: vectores de posición del *nodal point* de la estela en un tiempo anterior.
- rnpwknew: vectores de posición del *nodal point* de la estela en el tiempo actual.

4.3.2 *Capturas de los resultados arrojados por perf sobre el código de la versión original*

Figura 16: Análisis de perf para el código original.

4.3.3 *Tiempos*

Los valores mostrados en la tabla 2 se corresponden con el tiempo total de cada una de las simulaciones, y están expresados en *segundos*.
Flags: *default (-O0)*

Tiempos totales			
version	hardware	example 01	example large
original	Xeon E5-2680	9,77	16328,60 (04:32:08)
	i7-7660U	8,35	14386,43 (03:59:46)

Cuadro 2: Tiempos totales de la versión V0 en segundos.

Los valores de la tabla 3 se corresponden con la suma de todas la iteraciones, del tiempo de ejecución exclusivo de la función **convect** (2), y están expresados en *segundos*.

Tiempos de *convect*			
version	hardware	example 01	example large
original	Xeon E5-2680	9,25	15761,71 (04:22:41)
	i7-7660U	7,90	13883,70 (03:51:23)

Cuadro 3: Tiempos de la función *convect* en la versión V0 en segundos.

4.3.4 *Performance con roofline*

Los resultados presentados en esta sección, fueron tomados utilizando la herramienta Intel Advisor [23], más específicamente, el *feature* de *Roofline Analysis* [22], solo en el servidor *jupiterace*. Cabe destacar que los puntos en el gráfico se refieren a funciones (y ciclos dentro de funciones) específicos de cada simulación.

4.3.4.1 *Roofline para la versión V0: example 01*

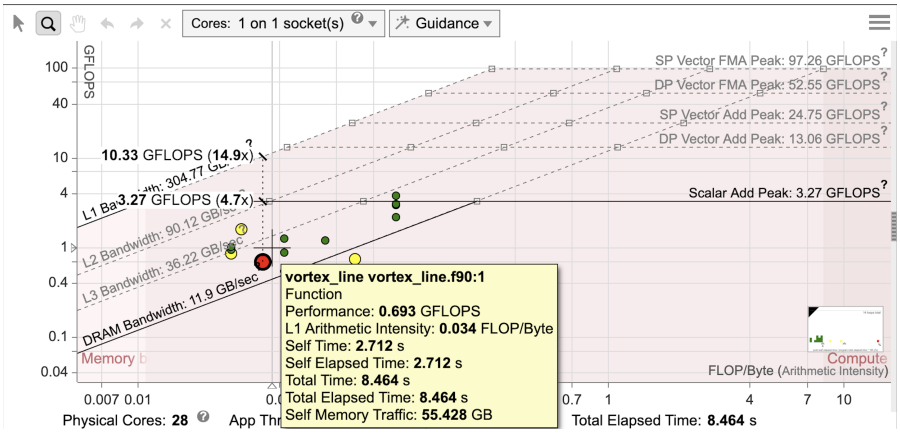

Figura 17: *Roofline* para la versión V0, example 01.

El orden de prioridades a la hora de optimizar es el intuitivo, es decir, es importante concentrase primero en el punto rojo, que corresponde a la función **vortex_line** 0,693 GFLOPS, luego en los amarillos, que pertenecen a dos *loops* dentro de **vortex_line** (0,741 GFLOPS y 0,863 GFLOPS) y a las llamadas de **cross_product** (1,601 GFLOPS) dentro de **vortex_line** y por último en los puntos verdes. En todos los casos es posible realizar mejoras en ambos ejes.

4.3.4.2 *Roofline para la versión V0: example large*

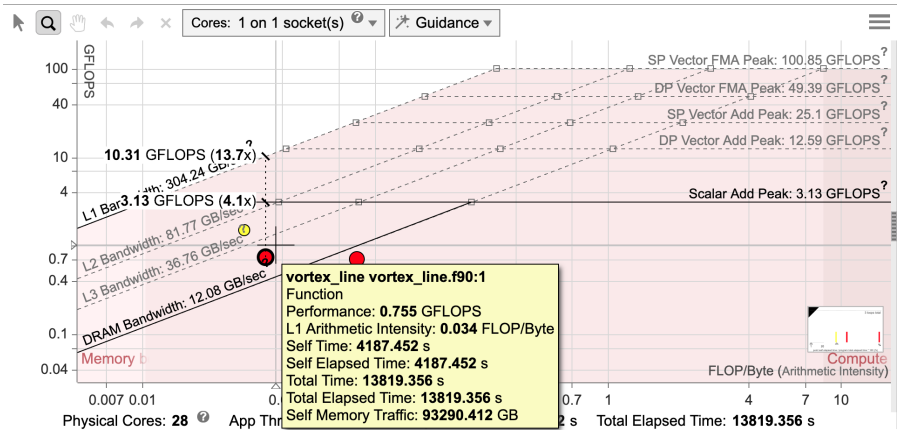

Figura 18: *Roofline* para la versión V0, example large.

Para el caso de esta simulación, se observa que el gráfico arroja tres puntos, dos de color rojo y uno de color amarillo. De acuerdo a la

performance se ubican, de menor a mayor, de la siguiente manera: un *loop* dentro de **vortex_line** (0,712 GFLOPS), seguido por la función **vortex_line** (0,755 GFLOPS) y finalmente, el punto de color amarillo que corresponde a la función **cross_product** (1,508 GFLOPS). Nuevamente, es posible realizar mejoras tanto en el cómputo, como en el manejo de la memoria.

4.4 v1: reescritura de secciones críticas en c con interfaz fortran

4.4.1 *Optimización*

Luego de identificar las partes críticas del código, se decidió migrarlas al lenguaje C. Esta decisión se fundó, tanto en el conocimiento del autor del trabajo en este lenguaje, como también de las facilidades que el mismo brinda en la integración con herramientas de HPC. Posteriormente, esta nueva versión fue integrada al código ya existente en lenguaje Fortran, mendiante el uso de interfaces provistas por el módulo **iso_c_binding**.

Sumado a la traducción, se realizaron algunas refactorizaciones de código como la transposición de las matrices involucradas en el cálculo de **convect**. Esto hizo que las celdas estuvieran contiguas al leerse desde memoria principal y se hiciera uso de la caché. Además, se cambió la forma de calcular la norma de los vectores, donde, en lugar de llamar a la función **dot_product** y luego calcular la raíz, se aprovechó que ya se iteraba sobre las coordenadas de los vectores y se utilizaron variables temporales para acumular la suma de las coordenadas. Finalmente, al concluir el ciclo, se le calculó la raíz a las variables temporales. Con estos cambios se logró mejorar levemente la *performance* de esta primera versión. Dicha mejora dió origen a dos nuevas funciones, **vortexLine** y **convect**, las cuales son leves mutaciones de las originales, mencionadas en la sección 4.3.

Luego de integrar el nuevo código, se procedió a la búsqueda de *flags* de compilación que potenciaron los rendimientos logrados hasta el momento. Mientras que en la versión original no se utilizaron *flags* de optimización, en esta nueva versión si. Los flags utilizados fueron: -O3 -ftree-vectorize -floop-block -funroll-loops.

4.4.2 *Tiempos*

Los valores mostrados en la tabla 4 se corresponden con el tiempo total de cada una de las simulaciones, y están expresados en *segundos*.

Los valores de la tabla 5 se corresponden con la suma de todas la iteraciones, del tiempo de ejecución exclusivo de la función **convect** (2), y están expresados en *segundos*.

Tiempos totales			
version	hardware	example 01	example large
original	Xeon E5-2680	9,77	16328,60 (04:32:08)
	i7-7660U	8,35	14386,43 (03:59:46)
traducción C	Xeon E5-2680	3,65	6074,49 (01:41:14)
	i7-7660U	3,73	4892,88 (01:21:32)

Cuadro 4: Tiempos totales de la versión V1 en segundos.

Tiempos de *convect*			
version	hardware	example 01	example large
original	Xeon E5-2680	9,25	15761,71 (04:22:41)
	i7-7660U	7,90	13883,70 (03:51:23)
traducción C	Xeon E5-2680	3,47	5894,34 (01:38:14)
	i7-7660U	3,55	4743,92 (01:19:03)

Cuadro 5: Tiempos de la función *convect* en la versión V1 en segundos.

4.4.3 *Performance con roofline*

Los resultados presentados en esta sección, fueron tomados utilizando la herramienta Intel Advisor [23], más específicamente, el *feature* de *Roofline Analysis* [22], solo en el servidor *jupiterace*. Cabe destacar que los puntos en el gráfico se refieren a funciones (y ciclos dentro de funciones) específicos de cada simulación.

4.4.3.1 *Roofline para la versión V1: example 01*

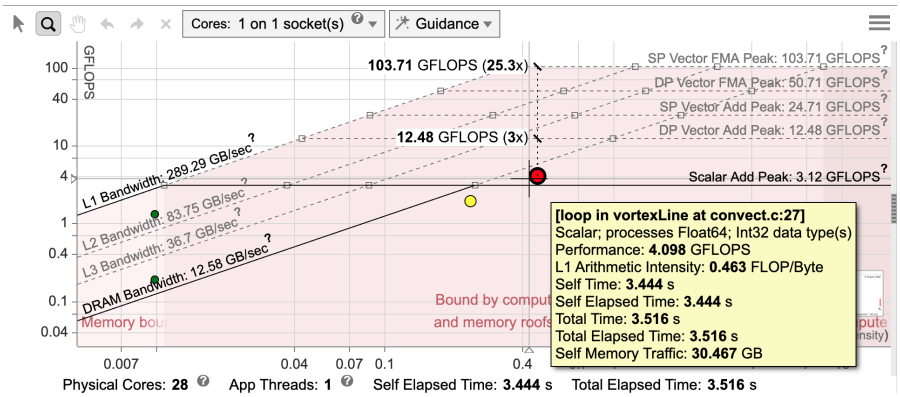

Figura 19: *Roofline* para la versión V1, example 01.

En esta versión se observa un solo punto rojo, que corresponde a un *loop* dentro de la nueva función **vortexLine**, del módulo de **convect**. Esta nueva función tiene una *performance* de 4,098 GFLOPS, donde la intensidad aritmética pasó de 0,034 FLOPS/byte en la versión anterior a 0,463 FLOPS/bytes en esta. Además del punto rojo de **vortexLine**,

existe un punto amarillo que pertenece a la función **vortex_line** original de Fortran, que corresponde a las llamadas por fuera de **convect**. Se destaca también, que si bien mejoró la *performance* total de la simulación, aún es posible realizar optimizaciones, tanto en el rendimiento como en el uso de memoria.

4.4.3.2 *Roofline para la versión V1: example large*

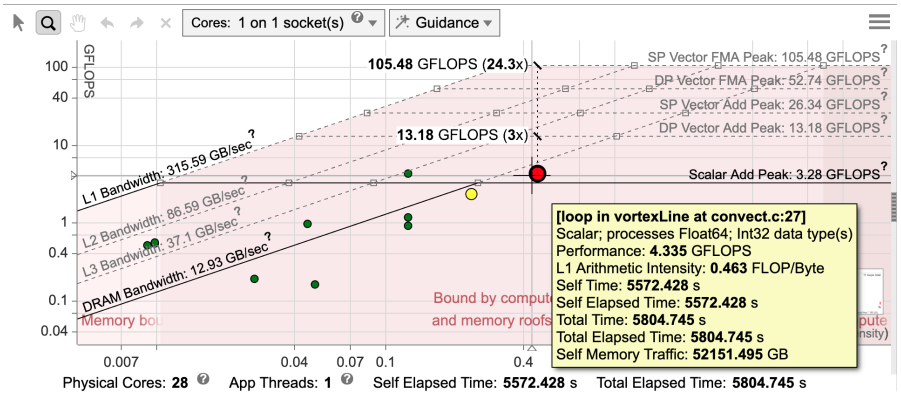

Figura 20: *Roofline* para la versión V1, example large.

Para esta simulación también se observa un punto de color rojo, correspondiente al *loop* dentro de la nueva versión **vortexLine**, con una *performance* de 4,335 GFLOPS. La intensidad aritmética de este punto paso de 0,083 FLOPS/byte en la versión anterior a 0,463 FLOPS/byte en esta. Por otro lado, se observa también un punto de color amarillo, que corresponde a la función **vortex_line** original de Fortran. Además, en esta versión se puede apreciar algunos puntos extra de color verdes, muy pequeños. Al igual que en la simulación del *example 01*, si bien se logró una mejora, aún es posible seguir optimizando el rendimiento y el uso de memoria.

4.5 V2: PARALELIZACIÓN DE CÓDIGO C UTILIZANDO OPENMP

4.5.1 *Optimización*

Luego de migrar las partes críticas del código al lenguaje C, se procedió a la paralelización del mismo, mediante el uso de la herramienta OpenMP (*Open Multi-Processing*).

La API de OpenMP admite la programación multiprocesamiento y multiplataforma de memoria compartida en una serie de lenguajes, arquitecturas de conjuntos de instrucciones y sistemas operativos. Consiste en un conjunto de directivas de compilación, bibliotecas de rutinas y variables de entorno, que influyen en el comportamiento en tiempo de ejecución. Utiliza un modelo portátil y escalable, que brinda a los programadores una interfaz simple y flexible para desarrollar aplicaciones paralelas.

La directiva ***parallel for*** encierra el código del ciclo, formando una región paralela, utilizando el modelo *fork-join*. Un hilo inicial que encuentra esta directiva, se bifurca, creando un equipo de hilos al

comienzo del ciclo (*fork*) y los une al final de la región (*join*). El hilo inicial se convierte en el hilo maestro del equipo, con un *thread ID* igual a cero, mientras que el resto se enumeran del 1 al número de hilos menos 1. A cada hilo de un equipo se le asigna una tarea implícita, que consiste de un conjunto de iteraciones del ciclo. Después de la finalización, el hilo maestro reanuda la ejecución de la tarea generadora. El número de subprocesos de una región paralela puede establecerse mediante la variable de entorno **OMP_NUM_THREADS**.

Por la naturaleza del problema, la directiva de la API de OpenMP utilizada fue la ya mencionada *parallel for*. La misma se utilizó dentro de la función **transpose**, por un lado y por otro lado en el ciclo principal de la función **convect**, donde los hilos se reparten las iteraciones del ciclo en partes iguales, un pedazo de tamaño fijo para cada hilo, y se asigna un round-robin para ejecutarse. El último pedazo puede ser más chico, debido al resto de la división. Además, el *hardware* disponible contaba con 4 hilos en la *macbook* y 28 en *jupiterace*. Debido a que no fué posible utilizar el 100 % de los recursos de *jupiterace*, la máxima cantidad de hilos utilizada fue 26. Mientras que en la macbook personal si se pudo aprovechar de los 4 hilos disponibles.

4.5.2 *Tiempos*

Los valores mostrados en la tabla 6 se corresponden con el tiempo total de cada una de las simulaciones, y están expresados en *segundos*.

Tiempos totales			
version	hardware	example 01	example large
original	Xeon E5-2680	9,77	16328,60 (04:32:08)
	i7-7660U	8,35	14386,43 (03:59:46)
traducción C	Xeon E5-2680	3,65	6074,49 (01:41:14)
	i7-7660U	3,73	4892,88 (01:21:32)
OMP	Xeon E5-2680	0,62	473,29 (00:07:53)
	i7-7660U	1,39	2394,02 (00:39:54)

Cuadro 6: Tiempos totales de la versión V2 en segundos.

Los valores de la tabla 7 se corresponden con la suma de todas la iteraciones, del tiempo de ejecución exclusivo de la función **convect** (2), y están expresados en *segundos*.

Tiempos de *convect*			
version	hardware	example 01	example large
original	Xeon E5-2680	9,25	15761,71 (04:22:41)
	i7-7660U	7,90	13883,70 (03:51:23)
traducción C	Xeon E5-2680	3,47	5894,34 (01:38:14)
	i7-7660U	3,55	4743,92 (01:19:03)
OMP	Xeon E5-2680	0,41	287,64 (00:04:47)
	i7-7660U	1,22	2246,08 (00:37:26)

Cuadro 7: Tiempos de la función *convect* en la versión V2 en segundos.

4.5.3 *Evolución de la velocidad y la escalabilidad fuerte*

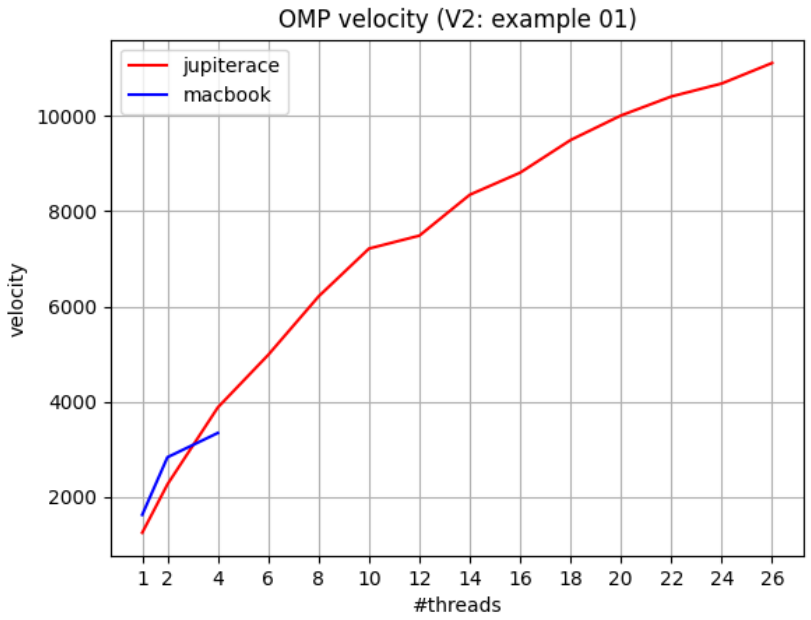

Figura 21: Velocidad en OMP, example 01.

En los gráficos 21 y 23, se puede apreciar una comparación entre la velocidad de actualización de una celda, en la matriz de vectores de posición del *nodal point* de la estela, versus la cantidad de *threads*. En ambos casos, se observa como al aumentar la cantidad de *threads* también aumenta la velocidad, logrando la mejor versión al utilizar la mayor cantidad de *threads* disponibles, es decir, 4 en la *macbook* y 26 en *jupiterace*.

Por otro lado, en los gráficos 22 y 24, se pueden observar los resultado de la escalabilidad fuerte en la actualización de una celda de la misma matriz. Donde se observa una disminución de la eficiencia a medida que se agregan más hilos al procesamiento, producto del *overhead* de comunicación y sincronización de los hilos.

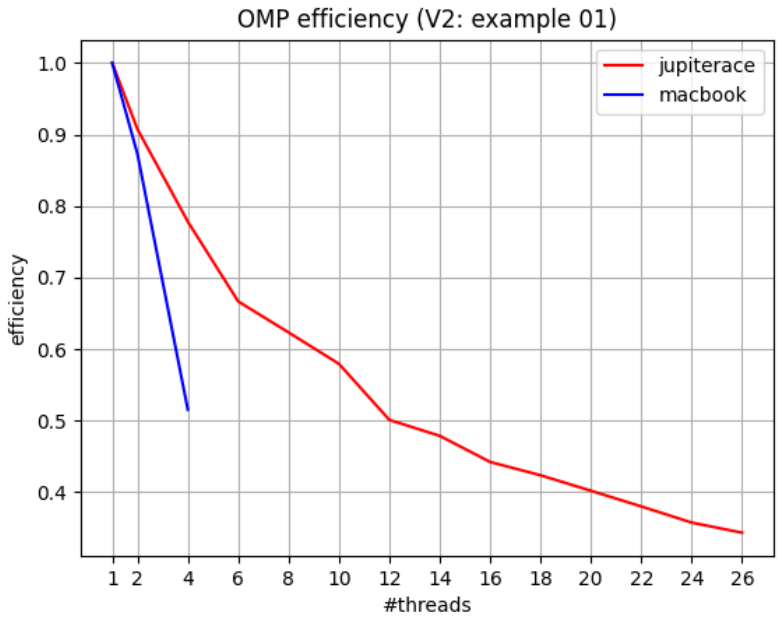

Figura 22: Eficiencia en OMP, example 01.

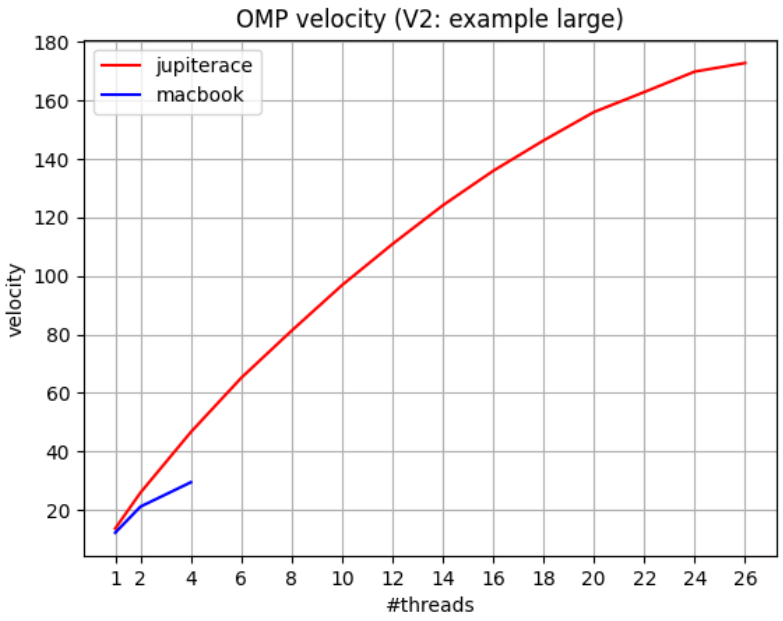

Figura 23: Velocidad en OMP, example large.

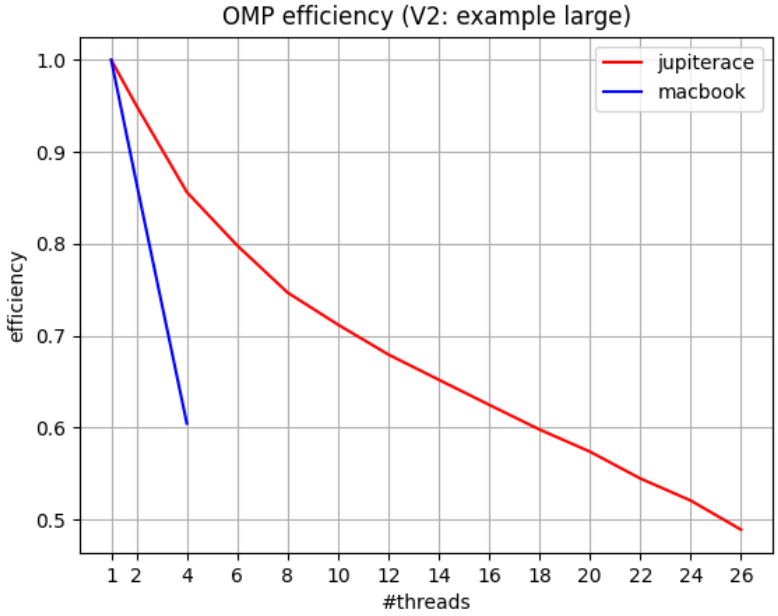

Figura 24: Eficiencia en OMP, example large.

4.5.4 *Performance con roofline*

Los resultados presentados en esta sección, fueron tomados utilizando la herramienta Intel Advisor [23], más específicamente, el *feature* de *Roofline Analysis* [22], solo en el servidor *jupiterace*. Cabe destacar que los puntos en el gráfico se refieren a funciones (y ciclos dentro de funciones) específicos de cada simulación.

4.5.4.1 *Roofline para la versión V2: example 01*

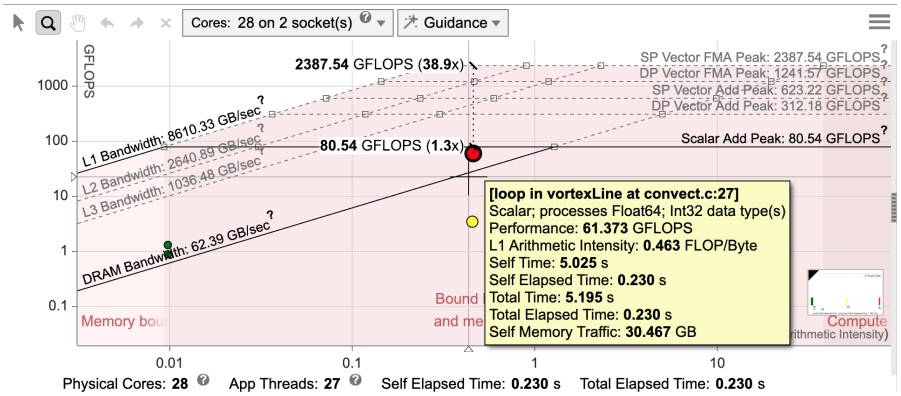

Figura 25: *Roofline* para la versión V2, example 01.

En esta nueva versión utilizando OMP, se observa como el paralelismo brindado por OMP, ayuda a inclementar la *performance*, haciendo que el *loop* de la función **vortexLine** que se veía en color rojo en la versión anterior, pase de 4,098 GFLOPS a 61,373 GFLOPS en esta nueva versión. A pesar de esta mejora, obtenida en su totalidad por medio del paralelismo que brinda OMP, aún sigue vigente la posibilidad

de mejorar la optimización, tanto en memoria como en rendimiento, como se mencionó en la versión anterior.

4.5.4.2 *Roofline para la versión V2: example large*

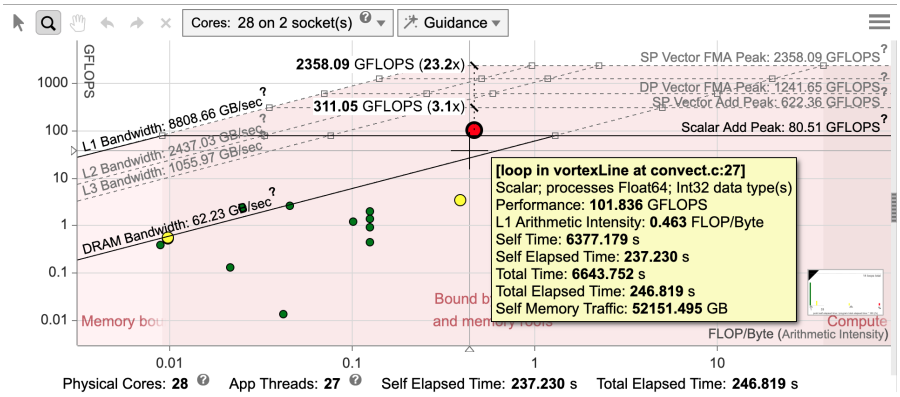

Figura 26: *Roofline* para la versión V2, example large.

Los resultados de esta simulación son similares a los del *example 01*, en donde se puede apreciar que el paralelismo brindado por OMP ayuda a mejorar la *performance* de la simulación, haciendo que pase de 4,335 GFLOPS a 101,836 GFLOPS en el *loop* de **vortexLine**. En esta nueva versión, comienzan aparecer nuevo puntos de color verde, que corresponden a funciones por fuera de *convect*. La conclusion es la misma, se logró una mejora importante, pero se puede seguir optimizando.

A pesar de que como se mencionó en 4.5.4, es posible seguir optimizando el código de esta versión, dado que aún no se alcanzó el *max peak* de *performance* posible en CPU, por limitaciones de tiempo se procederá directamente a realizar las versiones en GPU.

4.6 V3: CONVERSIÓN A CUDA DEL CÓDIGO C

4.6.1 *Optimización*

Finalizadas las pruebas con OpenMP, se retornó al código original, para comenzar la portación a CUDA.

Como se explicó en el capítulo 3, CUDA es una plataforma de propósito general de computación paralela y modelo de programación desarrollado por NVIDIA. Un desarrollo típico de CUDA se compone por uno o varios *kernels*, los cuales son funciones que se ejecutan N veces en paralelo por N hilos de diferentes, en lugar de solo una vez como las funciones normales de C++. Para el caso de este trabajo, se definió tan solo un *kernel* llamado *convect*, que como su nombre lo indica, es el equivalente a la función *convect* (2) original. Además de este *kernel*, se definieron algunas funciones *device*, como es el caso de *vortexLine*. Las funciones *device* son funciones que se ejecutan solamente en la GPU y que solo son llamadas por un *kernel*, en este caso *convect*.

El *kernel convect*, aprovecha los dos niveles de paralelismo explicados en 3, es decir, el paralelismo de datos de grano fino y paralelismo de hilos junto al paralelismo de datos de grano grueso y paralelismo de tareas, utilizando 512 hilos [1] por bloque y la cantidad necesaria de bloques para que la suma de todos los hilos de los bloques, alcance el valor de *nodal points* en la estela (*nnpwk*), es decir, 1515 para el *example 01* y 27555 para el *example large*. La distribución de las tareas se realiza mediante las siguientes ecuaciones:

- $\#\mathtt{threads} = 512$
- $\#\mathtt{blocks} = \lfloor (\mathtt{nnpwk} + \#\mathtt{threads} - 1)/\#\mathtt{threads} \rfloor$

Por lo tanto, los valores anteriores para cada simulación son:

- *example 01*: $\#\mathtt{threads} = 512,\ \#\mathtt{blocks} = 3$
- *example large*: $\#\mathtt{threads} = 512,\ \#\mathtt{blocks} = 54$

donde cada hilo realizar el trabajo de calcular un *nodal point* de la estela.

4.6.2 *Tiempos*

Los valores mostrados en la tabla 8 se corresponden con el tiempo total de cada una de las simulaciones, y están expresados en *segundos*.

Es importante remarcar que los valores obtenidos para la función **convect** en esta versión involucran un tiempo extra, correspondiente al copiado de los datos desde el *host* al *device*. Para ser justos con las versiones de GPU, y así realizar comparaciones reales entre CPU y GPU, el tiempo de copiado de datos no debe ser tenido en cuenta; es por esto que a continuación, en la tabla 9, se muestra la extensión de la tabla de la sección anterior, discriminando los tiempos de **convect** ahora en dos columnas; por un lado, se muestran los datos de la función de *host* (columna *function*), equivalente a la que se venía mostrando en las versiones anteriores y que incluye el tiempo extra de copiado, y por otro lado, el tiempo exclusivo de ejecución del *kernel* **convect** (columna *kernel*). Los datos se encuentran expresados en *segundos*.

1 No fué posible utilizar 1024 *threads* dado que se excedía la cantidad máxima de registros disponibles en ambos servidores.

Tiempos totales			
version	hardware	example 01	example large
original	Xeon E5-2680	9,77	16328,60 (04:32:08)
	i7-7660U	8,35	14386,43 (03:59:46)
traducción C	Xeon E5-2680	3,65	6074,49 (01:41:14)
	i7-7660U	3,73	4892,88 (01:21:32)
OMP	Xeon E5-2680	0,62	473,29 (00:07:53)
	i7-7660U	1,39	2394,02 (00:39:54)
CUDA 1	RTX 2080 Ti	3,51	451,43 (00:07:31)
	GTX 1070	5,29	645,50 (00:10:45)
	Tesla K40	1,71	787,84 (00:13:07)

Cuadro 8: Tiempos totales de la versión V3 en segundos.

Tiempos de *convect*					
version	hardware	example 01		example large	
original	Xeon E5-2680	9,25		15761,71 (04:22:41)	
	i7-7660U	7,90		13883,70 (03:51:23)	
traducción C	Xeon E5-2680	3,47		5894,34 (01:38:14)	
	i7-7660U	3,55		4743,92 (01:19:03)	
OMP	Xeon E5-2680	0,41		287,64 (00:04:47)	
	i7-7660U	1,22		2246,08 (00:37:26)	
		function	*kernel*	*function*	*kernel*
CUDA 1	RTX 2080 Ti	3,33	2,87	265,50	264,85
	GTX 1070	5,11	4,57	462,24	461,44
	Tesla K40	1,10	0,44	108,75	107,74

Cuadro 9: Tiempos del *kernel convect* en la versión V3 en segundos.

4.6.3 *Evolución de la velocidad*

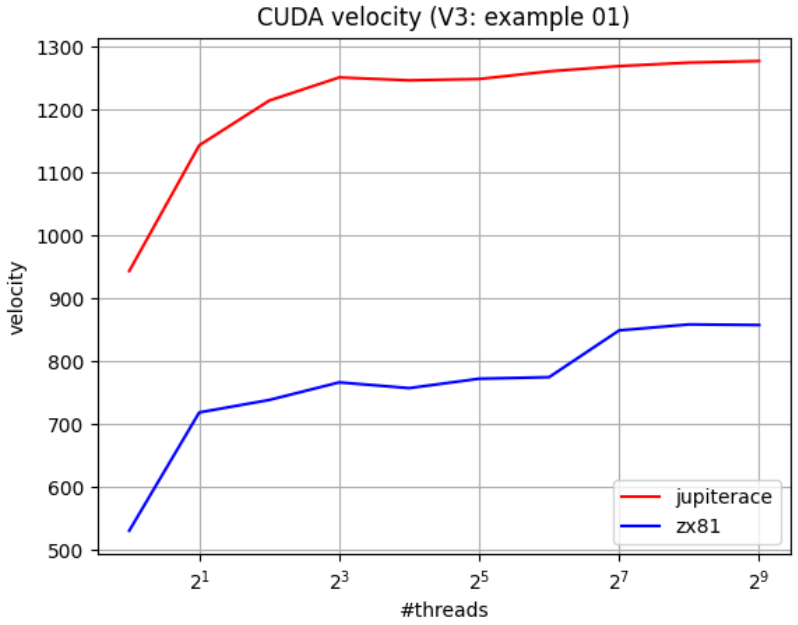

Figura 27: Velocidad en CUDA (V3), example 01.

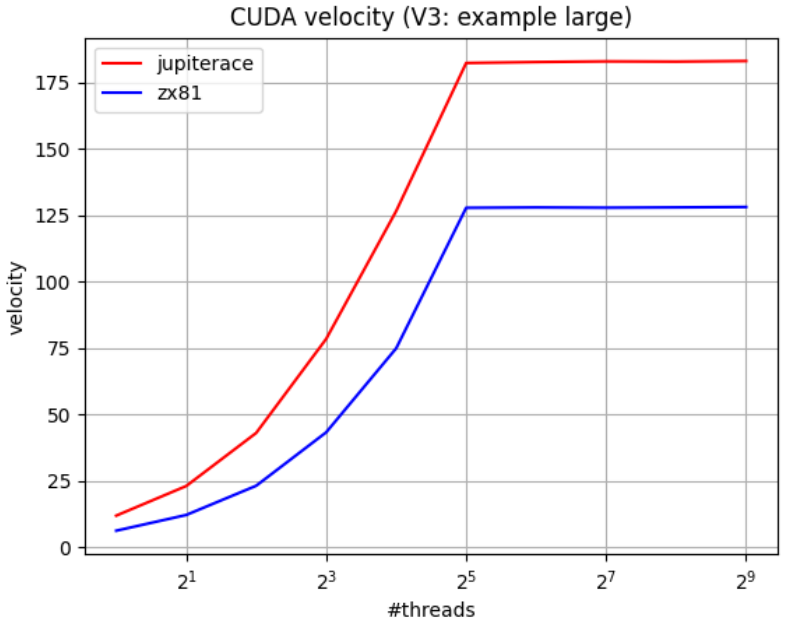

Figura 28: Velocidad en CUDA (V3), example large.

Para el **example 01** se puede observar que con poca cantidad de *threads*, la velocidad promedio de actualización de una celda de la matriz *rnpwk* crece rápidamente entre 1 y 8 *threads* en **zx81**, logrando una constante entre los 8 y 512 *threads*. El comportamiento es similar en **jupiterace**, con la diferencia de que la parte de crecimiento se mantiene entre 1 y 16 *threads*, siendo agresivo en el cambio de 1 a 2

threads y manteniendo un crecimiento sostenido hasta los 16. Al igual que en **zx81**, el resultado continua constante hasta el final.

Para el **example large** se observa una curva de velocidad similar entre ambas máquinas, a pesar de la gran diferencia numérica existente. Se visualiza un creciemiento exponencial, más pronunciado en **jupiterace**, al comienzo de la curva, entre 1 y 32 *threads*. Luego de los 32 *threads*, se mantiene constante hasta el final, en ambos casos.

4.6.4 *Performance con roofline*

Los resultados presentados en esta sección, fueron tomados utilizando la herramienta NVIDIA Nsight Compute [37], más específicamente, el *feature* de *Roofline Charts* [38], solo en el servidor *jupiterace*. Cabe destacar que el código que involucra a la función **convect**, y por ende la porción traducida a CUDA en esta versión, solo utiliza el tipo de dato **double** (64 bits), por lo tanto, la *performance* de interés para analizar en esta sección, es la de precisión doble (punto de color rojo en el gráfico) y no la simple (punto de color morado).

4.6.4.1 *Roofline para la versión V3: example 01*

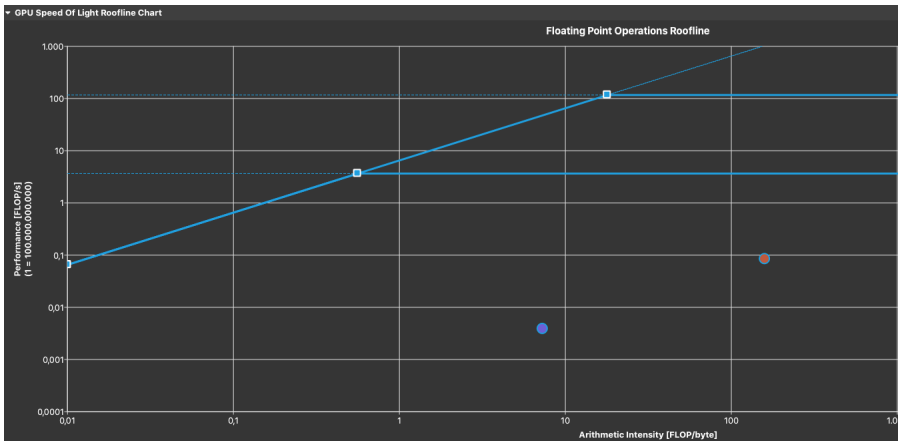

Figura 29: *Roofline* para la versión V3, example 01.

Con 8,3 GFLOPS de *performance* y 159 FLOPS/byte de AI, una mejora de 329X con respecto a la AI de la versión anterior (0,483 FLOPS/byte). Aún así, se observa una *performance* alejada de ambos techos del gráfico de *roofline*, tanto de la memoria como del cómputo. Se debe revisar el *kernel* para aumentar su *performance*, sobretodo buscando explotar el ancho de banda de memoria, haciendo uso de las memorias de acceso rápido que provee la GPU.

4.6.4.2 *Roofline para la versión V3: example large*

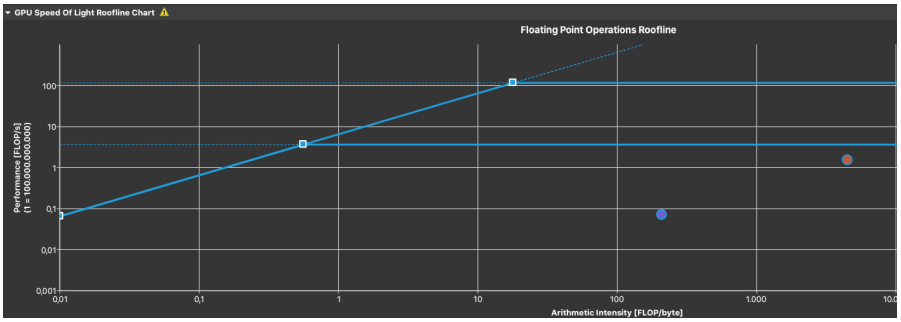

Figura 30: *Roofline* para la versión V3, example large.

En esta simulación, se observa un resultado más alentador. Con 150 GFLOPS de *performance* y 4522 FLOPS/byte de AI (0,463 en la versión anterior), el valor logrando se encuentra más cercano al techo de *performance*. A pesar de esto, aún quedan optimizaciones por realizar; se debe seguir las mismas sugerencias que para el *example 01*.

4.7 V4: *kernel* ALTERNATIVO EN CUDA

4.7.1 *Optimización*

Por último, luego de traducir el código original a CUDA, se realizó una prueba con un *kernel* alternativo. Este nuevo *kernel* explota aún más el paralelismo de grano grueso, utilizando un bloque por cada *nodal point* en la estela (*nnpwk*), e iterando los *threads* de cada bloque sobre el número total de *vortex lines* en la superficie sustentadora (*nvlls*) por un lado; lo que sería el equivalente al ciclo de la línea 4 en el algoritmo ***convect*** (2) original de la sección 4.3.1, y por otro, la resta entre el número actual de *vortex lines* en la estela (*nvlwkt*) y el número de *vortex lines* a convectar (*nvlcv*), es decir, $nvlwkt - nvlcv$; lo que sería el equivalente al ciclo de la línea 8 en el algoritmo ***convect*** (2) original. Nuevamente, la cantidad de *threads* utilizada fué 512, dado que el *kernel* excedía la cantidad máxima de registros. Por lo tanto, la distribución de las tareas se realiza mediante las siguientes ecuaciones:

- $\#threads = 512$
- $\#blocks = nnpwk$

Entonces, los valores anteriores para cada simulación son:

- *example 01*: $\#threads = 512,\ \#blocks = 1515$
- *example large*: $\#threads = 512,\ \#blocks = 27555$

Para este nuevo kernel, se utilizó una librería muy conocida en el mundo de CUDA, llamada CUB. Esta librería proporciona componentes de software reutilizables de última generación, para cada capa del modelo de programación CUDA. La clase **BlockReduce** provee

métodos colectivos para calcular una reducción paralela de elementos divididos en un bloque de subprocesos CUDA. En este trabajo, se utilizó la clase *BlockReduce* para calcular una suma interbloques, de valores acumulados por los hilos de cada bloque, es decir, dentro de un bloque los hilos computan valores temporales, y esos valores luego se juntan con los equivalentes de los otros bloques, conformando un resultado final del *kernel*.

4.7.2 *Tiempos*

Los valores mostrados en la tabla 10 se corresponden con el tiempo total de cada una de las simulaciones, y están expresados en *segundos*.

Tiempos totales			
version	hardware	example 01	example large
original	Xeon E5-2680	9,77	16328,60 (04:32:08)
	i7-7660U	8,35	14386,43 (03:59:46)
traducción C	Xeon E5-2680	3,65	6074,49 (01:41:14)
	i7-7660U	3,73	4892,88 (01:21:32)
OMP	Xeon E5-2680	0,62	473,29 (00:07:53)
	i7-7660U	1,39	2394,02 (00:39:54)
CUDA 1	RTX 2080 Ti	3,51	451,43 (00:07:31)
	GTX 1070	5,29	645,50 (00:10:45)
	Tesla K40	1,71	787,84 (00:13:07)
CUDA 2	RTX 2080 Ti	0,76	406,35 (00:06:46)
	GTX 1070	1,03	633,97 (00:10:33)
	Tesla K40	1,26	866,52 (00:14:26)

Cuadro 10: Tiempos totales de la versión V4 en segundos.

Al igual que en la versión anterior, es necesario realizar la diferenciación entre el tiempo total de ejecución de la función **convect**, y el tiempo de ejecución del *kernel*. Nuevamente, los valores de la tabla 11 están expresados en *segundos*.

Tiempos de *convect*					
version	hardware	example 01		example large	
original	Xeon E5-2680	9,25		15761,71 (04:22:41)	
	i7-7660U	7,90		13883,70 (03:51:23)	
traducción C	Xeon E5-2680	3,47		5894,34 (01:38:14)	
	i7-7660U	3,55		4743,92 (01:19:03)	
OMP	Xeon E5-2680	0,41		287,64 (00:04:47)	
	i7-7660U	1,22		2246,08 (00:37:26)	
		function	*kernel*	*function*	*kernel*
CUDA 1	RTX 2080 Ti	3,33	2,87	265,50	264,85
	GTX 1070	5,11	4,57	462,24	461,44
	Tesla K40	1,10	0,44	108,75	107,74
CUDA 2	RTX 2080 Ti	0,58	0,18	221,30	220,60
	GTX 1070	0,84	0,35	451,58	450,85
	Tesla K40	0,71	0,14	191,10	189,64

Cuadro 11: Tiempos del *kernel convect* en la versión V4 en segundos.

4.7.3 *Evolución de la velocidad*

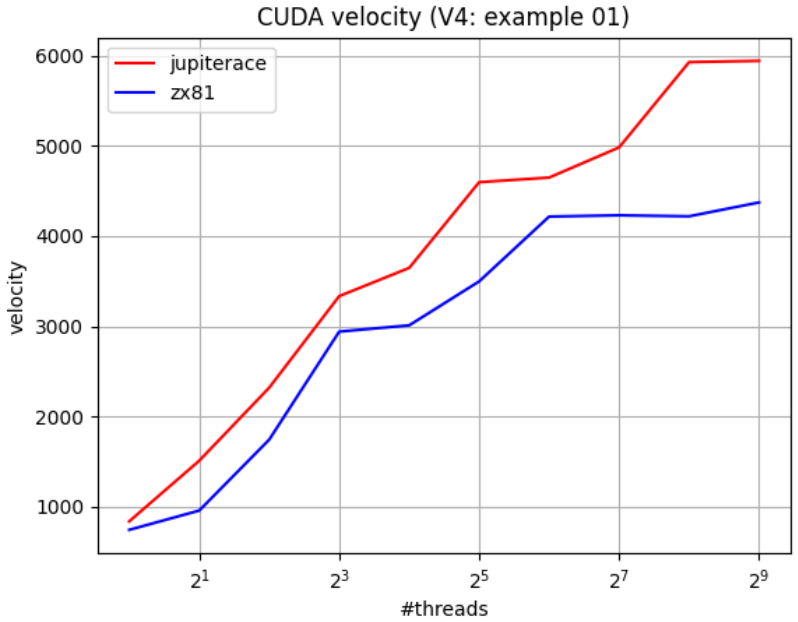

Figura 31: Velocidad en CUDA (V4), example 01.

Para la simulación **example 01** de esta versión, se observa en ambos casos un comienzo con un crecimiento exponencial, seguido de una constante. La diferencia entre ambos servidores radica en dos puntos, por un lado una gran diferencia numérica existente entre los servidores.

Pero además, la parte inicial de crecimiento exponencial se mantiene entre 1 y 8 *threads* en **zx81**; mientras que en **jupiterace** se prolonga de manera más agresiva hasta los 32 *threads*.

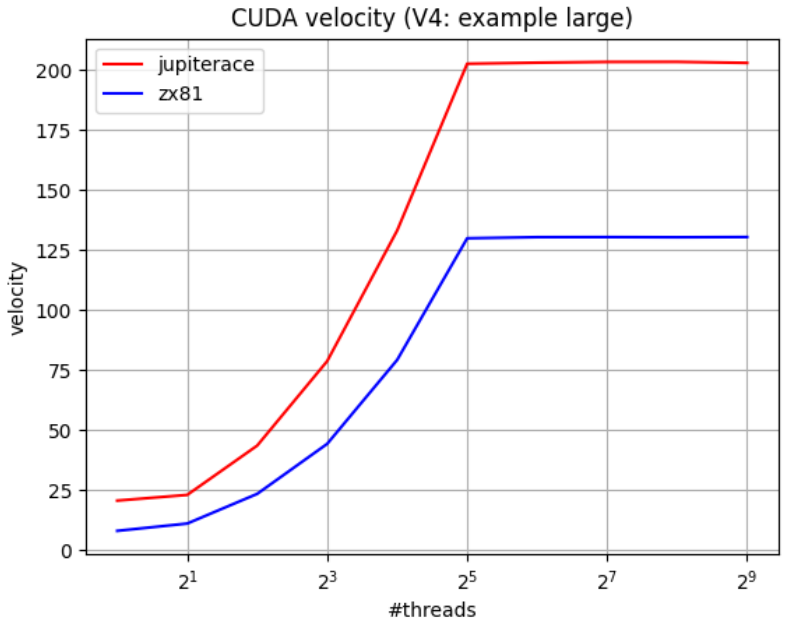

Figura 32: Velocidad en CUDA (V4), example large.

Para la simulación **example large** de esta versión, se observan resultados muy similares en los servidores *zx81* y *jupiterace* a pesar de la gran diferencia númerica existente entre ambos, donde las corridas desde 1 a 32 *threads* muestran un creciemiento exponencial, que luego se transforma en una constante hasta los 512 *threads*.

4.7.4 *Performance con roofline*

Los resultados presentados en esta sección, fueron tomados utilizando la herramienta NVIDIA Nsight Compute [37], más específicamente, el *feature* de *Roofline Charts* [38], solo en el servidor *jupiterace*. En esta versión también es reelevante la aclaración realizada en la sección anterior, solo el punto rojo (precisión doble) es el de interés.

4.7.4.1 *Roofline para la versión V4: example 01*

Con un resultado de 122 GFLOPS, se observa un valor logrado cercano al techo de *perfomance*. Aún así, es posible seguir optimizando para lograr mejores resultados, dado que aún no se ha alcanzado la *performance* pico. Las recomendaciones son las mismas que las de la versión anterior.

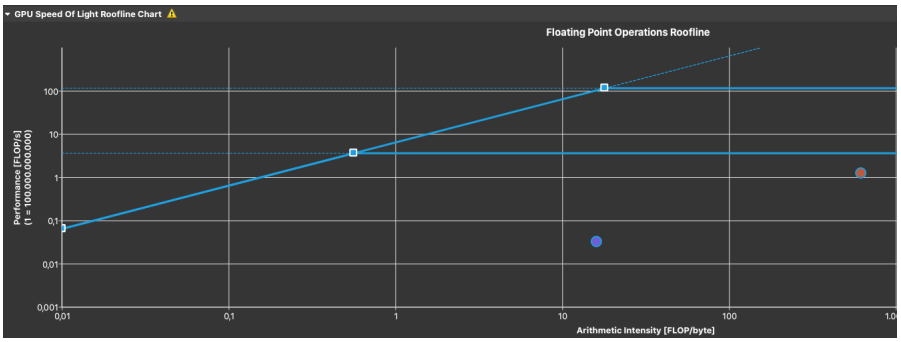

Figura 33: *Roofline* para la versión V4, example 01.

4.7.4.2 *Roofline para la versión V4: example large*

En esta versión se observa un resultado similar al del *example 01*, con una *performance* de 179 GFLOPS. De igual manera, esta versión comparte la conclusión con el *example 01*, es decir, aún es posible seguir optimizando.

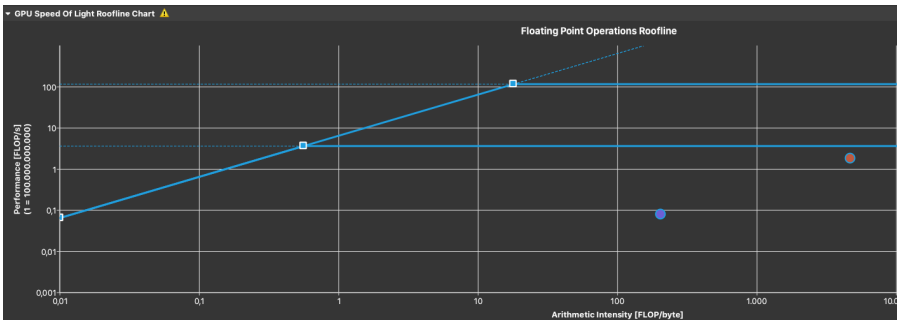

Figura 34: *Roofline* para la versión V4, example large.

5 CONCLUSIONES Y TRABAJO FUTURO

En este trabajo final se presentó una optimización al Método de Red de Vórtices Inestacionario, desarrollado por el Dr. Preidikman. Para lograr el objetivo, se utilizaron distintas técnicas de Computación de Alto Desempeño y Computación Paralela; pasando por algunas mejoras en la arquitectura tradicional de CPU, mediante la refactorización del código existente, el uso de *flags* de compilación y de la utilización de herramientas y librerías como OpenMP.

Luego de las optimizaciones realizadas en CPU, se procedió a la migración del código a CUDA, la plataforma de propósito general de computación paralela desarrollada por NVIDIA. Se crearon dos implementaciones distintas, cada una con una granularidad de paralelismo particular.

Como dijo Bill Dally en [10]: "All performance is from parallelism" (Toda *performance* viene del paralelismo); este proyecto es una fiel demostración de esta frase. A lo largo de este trabajo final se puede apreciar, como la incorporación de procesos paralelos, es lo que efectivamente logra mejoras en la *performance*.

Si bien por limitaciones de tiempo no fué posible llevar cada una de las distintas versiones al máximo desempeño posible, se logró un reducción importante en los tiempos de cada simulación. Para el **example 01**, se logró mejorar el tiempo en un $93,7\,\%$, tanto el tiempo total de la simulación como el tiempo exclusivo de la función *convect*, la cual fué el punto central de todas las optimizaciones. Esta mejora equivale a un 15 X de ganancia. Este resultado se logró en la versión que utilizaba OMP para correr en CPU. Por el lado de la simulación **example large**, se obtuvieron resultados distintos respecto al tiempo total versus la función *convect*. La mejor versión para esta simulación se alcanzó con el *kernel* de CUDA que utilizaba la lib CUB, logrando una mejora del $97,6\,\%$ en el tiempo total de la simulación (40 X). Mientras que en la función **convect** se logró una mejora del $98,6\,\%$, lo que equivale a una optimización de 71 X.

Como se mencionó en el capítulo 4 aún es posible realizar optimizaciones para incrementar la *performance* de cada *kernel*, ya que el hardware no ha sido una limitación. El cambio más importante a realizar es la forma en que se leen los datos de memoria global, es necesario aumentar el ancho de banda, utilizando las memorias rápidas que provee la GPU, como por ejemplo *shared memory*. Actualmente, cada hilo accede a memoria principal para buscar solo 24 bytes contiguos; idealmente se debería operar en bloque y luego repartir el trabajo a los hilos.

Más allá de las mejoras puntuales para los *kernels* ya desarrollados, es posible realizar algunos expermientos a futuro para incrementar aún más el rendimiento. Utilizar el tipo de dato *float* (32 bits) en lugar de *double* (64 bits), si la precisión de las simulaciones así lo

permite, es un experimento para tener en cuenta; las GPU brindan una *performance* mayor al utilizar este tipo de dato. En el caso de la **GeForce RTX 2080 Ti** de **jupiterace**, la *performance* FP32 es de 13,45 TFLOPS versus 420,2 GFLOPS de FP64. De hecho, al calcular *roofline* con Nsight Compute [38], la herramienta arrojó este cambio como sugerencia: "La relación de rendimiento máximo de *float* (fp32) a *double* (fp64) en este dispositivo es 32:1.".

Por otro lado, los resultados obtenidos con la GPU **Tesla K40** del servidor *yopi* de FaMAF, muestran una mejoría aún mayor en los tiempos del *kernel*, no así con la simulación completa. Este resultado se debe a que esta placa está diseñada para cálculo, con una *performance* FP64 de 1,4 TFLOPS, lo cual hace que el tiempo del *kernel* sea pequeño, pero que está acompañada por un hardware limitado, lo que impide la reducción del tiempo total de la simulación. Es por esto que es importante realizar pruebas en algún servidor no tan limitado, que contenga una GPU diseñada para cálculo.

Por último, otro experimento interesante sería mover toda la simulación a la GPU. Recordemos que actualmente la función **convect** es la única que utiliza la GPU, mientras que el resto del código de la simulación corre en CPU. Es decir, ahora cada paso de la simulación procesa datos en la CPU, copia esos datos a la GPU para seguir procesandolos allí (en la parte correspondiente a la función **convect**) y al finalizar vuelve a la CPU, escribiendo los resultados en distintos archivos. Lo que se propone con este experimento es que el proceso completo, es decir, todos los pasos de una simulación, se realicen en GPU; copiando los datos necesarios al principio, realizando todo el cálculo necesario en GPU y finalmente copiando nuevamente al *host* los resultados para su posterior informe.

BIBLIOGRAFÍA

[1] SA Ansari, R Żbikowski y K Knowles. «Non-linear unsteady aerodynamic model for insect-like flapping wings in the hover. Part 1: methodology and analysis». En: *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering* 220.2 (2006), págs. 61-83.

[2] Salman A Ansari, R Żbikowski y Kevin Knowles. «Non-linear unsteady aerodynamic model for insect-like flapping wings in the hover. Part 2: implementation and validation». En: *Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering* 220.3 (2006), págs. 169-186.

[3] EH Atta, OA Kandil, DT Mook y AH Nayfeh. «Unsteady aerodynamic loads on arbitrary wings including wing-tip and leading-edge separation». En: *AIAA paper* 156 (1977), pág. 1977.

[4] SM Belotserkovskii. «Study of the unsteady aerodynamics of lifting surfaces using the computer». En: *Annual Review of Fluid Mechanics* 9.1 (1977), págs. 469-494.

[5] *CS Roofline Toolkit*. URL: https://bitbucket.org/berkeleylab/cs-roofline-toolkit/src/master/.

[6] S.J. Chapman. *FORTRAN FOR SCIENTISTS & ENGINEERS*. McGraw-Hill Education, 2017. ISBN: 9780073385891. URL: https://books.google.com.ar/books?id=OQhBMQAACAAJ.

[7] J. Cheng, M. Grossman y T. McKercher. *Professional CUDA C Programming*. EBL-Schweitzer. Wiley, 2014. ISBN: 9781118739327. URL: https://books.google.com.ar/books?id=q3DvBQAAQBAJ.

[8] *Cmake*. URL: https://cmake.org.

[9] NVIDIA Coorporation. «Nvidia turing GPU architecture: Graphics Reinvented». En: *Tech. Rep. WP-09183-001 v01* (2018).

[10] W Dally. «Efficiency and programmability: Enablers for exascale». En: *SC13* (2013).

[11] Michael H Dickinson y Karl G Gotz. «Unsteady aerodynamic performance of model wings at low Reynolds numbers». En: *Journal of experimental biology* 174.1 (1993), págs. 45-64.

[12] Michael H Dickinson, Fritz-Olaf Lehmann y Sanjay P Sane. «Wing rotation and the aerodynamic basis of insect flight». En: *Science* 284.5422 (1999), págs. 1954-1960.

[13] Jack J Dongarra, Piotr Luszczek y Antoine Petitet. «The LINPACK benchmark: past, present and future». En: *Concurrency and Computation: practice and experience* 15.9 (2003), págs. 803-820.

[14] Kevin Dowd y Charles Severance. «High performance computing». En: (2010).

[15] F Edward Ehlers. *A method for computing the leading-edge suction in a higher-order panel method*. NASA, 1984.

[16] Victor Eijkhout. *Introduction to High Performance Scientific Computing*. The MIT Press, 2011.

[17] Charles P Ellington, Coen Van Den Berg, Alexander P Willmott y Adrian LR Thomas. «Leading-edge vortices in insect flight». En: *Nature* 384.6610 (1996), págs. 626-630.

[18] Charles Porter Ellington. «The aerodynamics of hovering insect flight. IV. Aerodynamic mechanisms». En: *Philosophical Transactions of the Royal Society of London. B, Biological Sciences* 305.1122 (1984), págs. 79-113.

[19] Gabriela Gonzalez, Stephen Fairhurst, P Ajith, Patrick Brady, Alessandra Buonanno, Alessandra Corsi, Peter Fritschel, Bala Iyer, Joey Key, Sergey Klimenko y col. «The LIGO Scientific Collaboration». En: *Bulletin of the American Physical Society* 60 (2015).

[20] B. Gregg. *Systems Performance: Enterprise and the Cloud*. Always learning. Prentice Hall, 2013. ISBN: 9780133390094. URL: https://books.google.com.ar/books?id=xQdvAQAAQBAJ.

[21] John L. Hennessy y David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055.

[22] *Intel Advisor: Roofline Analysis*. URL: https://software.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top/survey-trip-counts-flops-and-roofline-analyses/roofline-analysis.html.

[23] *Intel Advisor*. URL: https://software.intel.com/content/www/us/en/develop/tools/advisor.html.

[24] *Intel VTune*. URL: https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.

[25] OA Kandil, DT Mook y AH Nayfeh. «Nonlinear prediction of aerodynamic loads on lifting surfaces». En: *Journal of Aircraft* 13.1 (1976), págs. 22-28.

[26] Th H von Karman y William R Sears. «Airfoil theory for non-uniform motion». En: *Journal of the Aeronautical Sciences* 5.10 (1938), págs. 379-390.

[27] Joseph Katz. «Lateral aerodynamics of delta wings with leading-edge separation». En: *AIAA journal* 22.3 (1984), págs. 323-328.

[28] Joseph Katz y Allen Plotkin. *Low-speed aerodynamics*. Vol. 13. Cambridge university press, 2001.

[29] P Konstadinopoulos, D Mook y A NAYFEH. «A numerical method for general, unsteady aerodynamics». En: *7th Atmospheric Flight Mechanics Conference*. 1981, pág. 1877.

[30] P Konstadinopoulos, DF Thrasher, DT Mook, AH Nayfeh y L Watson. «A vortex-lattice method for general, unsteady aerodynamics». En: *Journal of aircraft* 22.1 (1985), págs. 43-49.

[31] Ryan J. Leng. *Simplified Computer Memory Hierarchy*. 2020. URL: https://sites.google.com/site/cachememory2011/memory-hierarchy.

[32] Annie Leroy, Franck Buron y Philippe Devinant. *Unsteady aerodynamic model for thin wings with evolutive vortex sheets*. Inf. téc. NATO RESEARCH y TECHNOLOGY ORGANIZATION NEUILLY-SUR-SEINE (FRANCE), 2003.

[33] MJ Lighthill. «A new approach to thin aerofoil theory». En: *The Aeronautical Quarterly* 3.3 (1951), págs. 193-210.

[34] Therese A Markow y Patrick O'Grady. *Drosophila: a guide to species identification and use*. Elsevier, 2005.

[35] DT Mook y SA Maddox. «Extension of a Vortex-Lattice Method to Include the Effects ofLeading-Edge Separation». En: *Journal of Aircraft* 11.2 (1974), págs. 127-128.

[36] NVIDIA, Péter Vingelmann y Frank H.P. Fitzek. *CUDA, release: 11.0*. 2020. URL: https://developer.nvidia.com/cuda-toolkit.

[37] *Nvidia Nsight Compute: Profiling Tool*. URL: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.

[38] *Nvidia Nsight Compute: Roofline Analysis*. URL: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#roofline.

[39] *Nvidia Nvprof*. URL: https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[40] David A Patterson y John L Hennessy. «Computer Organization and Design: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)». En: *Paperback, Morgan Kaufmann Publishers* (2013).

[41] S. Preidikman, B. A. Roccia, M. L. Verstraete, L. R. Ceballos y B." Balachandran. «A Computational Aeroelastic Framework for Studying Non-conventional Aeronautical Systems». En: *Multibody Mechatronic Systems*. Ed. por João Carlos Mendes Carvalho, Daniel Martins, Roberto Simoni y Henrique Simas. Cham: Springer International Publishing, 2018, págs. 325-334. ISBN: 978-3-319-67567-.

[42] Sergio Preidikman. «Numerical simulations of interactions among aerodynamics, structural dynamics, and control systems». Tesis doct. Virginia Tech, 1998.

[43] Sergio Preidikman y Dean Mook. «Time-domain simulations of linear and nonlinear aeroelastic behavior». En: *Journal of Vibration and Control* 6.8 (2000), págs. 1135-1175.

[44] Ravi Ramamurti y William C Sandberg. «A three-dimensional computational study of the aerodynamic mechanisms of insect flight». En: *Journal of experimental biology* 205.10 (2002), págs. 1507-1518.

[45] C Rehbach. «Numerical calculation of three-dimensional unsteady flows with vortex sheets». En: *16th Aerospace Sciences Meeting*. 1978, pág. 111.

[46] Bruno A. Roccia, Sergio Preidikman, Julio C. Massa y Dean T. Mook. «Modified Unsteady Vortex-Lattice Method to Study Flapping Wings in Hover Flight». En: *AIAA Journal* 51.11 (2013), págs. 2628-2642. DOI: 10.2514/1.J052262. eprint: https://doi.org/10.2514/1.J052262. URL: https://doi.org/10.2514/1.J052262.

[47] Akira Sakurai e Hiroshi Uchihori. «Unsteady vortex lattice calculation of the flow around a slender delta wing». En: *The Institute of Space and Astronautical Science Report* 12 (1990), págs. 53-59.

[48] Sanjay P Sane. «The aerodynamics of insect flight». En: *Journal of experimental biology* 206.23 (2003), págs. 4191-4208.

[49] T. Sterling, M. Brodowicz y M. Anderson. *High Performance Computing: Modern Systems and Practices*. Elsevier Science, 2017. ISBN: 9780124202153. URL: https://books.google.com.ar/books?id=qOHIBAAAQBAJ.

[50] Coen Van Den Berg y Charles P Ellington. «The three–dimensional leading–edge vortex of a 'hovering'model hawkmoth». En: *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences* 352.1351 (1997), págs. 329-340.

[51] Michael S Vest y Joseph Katz. «Unsteady aerodynamic model of flapping wings». En: *AIAA journal* 34.7 (1996), págs. 1435-1440.

[52] Samuel Williams. «Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore». En: *ACM Communications* (2009).

[53] Charlene Yang. «Hierarchical Roofline Analysis: How to Collect Data using Performance Tools on Intel CPUs and NVIDIA GPUs». En: *arXiv preprint arXiv:2009.02449* (2020).

[54] *perf: Linux profiling with performance counters*. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

Los abajo firmantes, miembros del Tribunal de evaluación de tesis, damos fe que el presente ejemplar impreso se corresponde con el aprobado por este Tribunal.