UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ACATLÁN

ADAPTACIÓN DE UN CÓDIGO NUMÉRICO DE DINÁMICA DE FLUIDOS EN OPEN MP AL MODELO DE ACELERACIÓN OPEN ACC

TESIS

QUE PARA OBTENER EL TITULO DE

LIC. EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN

PRESENTA

JIMENA ALDAPE RAMÍREZ

ASESOR: DR. CARLOS COUDER CASTAÑEDA

FECHA: FEBRERO 2016

SANTA CRUZ ACATLÁN, NAUCALPAN, ESTADO DE MÉXICO

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Jimena Aldape Ramírez, *Adaptación de un código numérico de dinámica de fluidos en OpenMP al modelo de aceleración de OpenACC,* Tesis para obtener el título de Lic. en Matemáticas Aplicadas y Computación

.

Dedicatoria

Dedico este trabajo a mis padres Maricela y Francisco por la confianza para dejarme elegir mi profesión y el apoyo que me brindaron para concluir esta etapa.

A la Universidad Nacional Autónoma de México por poner a mi alcance los recursos necesarios para concluir mi licenciatura y a todos los docentes que hacen de ella una de las mejores universidades.

También está dedicada a mis hermanas Renata y Jessica, a mi compañero de vida David y finalmente a todos mis amigos que estuvieron conmigo compartiendo momentos inolvidables dentro de las aulas.

Resumen

Esta tesis la constituyen el diseño y la recodificación de un programa paralelo escrito en FORTRAN 2003 que sirve para simular flujo supersónico en eyectores con directivas OpenMP al modelo de directivas OpenACC, con la finalidad de que se pueda ejecutar en las nuevas tarjetas de procesamiento gráfico NVIDIA con la finalidad de obtener un mayor rendimiento del cómputo comparado con una máquina multi-núcleo a un menor costo de hardware y un esfuerzo de programación similar a OpenMP.

La necesidad de cómputo cada vez más creciente de las simulaciones en computadora trajo la creación de la computadora paralela, dicha necesidad en un principio fue suplida a través de multi-computadoras que actualmente se conocen como *clusters*, posteriormente la creación y el abaratamiento de los sistemas multi-núcleo llevó a construir los *clusters* de forma híbrida, es decir, compuestos de sistemas multi-núcleo. A partir de la aparición de las tarjetas de procesamiento gráfico cada componente del cluster o cada nodo le fue añadido más poder de cómputo por medio de dichas tarjetas, también reduciendo el consumo de energía, ya que las tarjetas de procesamiento gráfico pueden dar un rendimiento similar o mayor de un sistema multi-núcleo.

No obstante, a pesar de que las tarjetas gráficas ofrecen un poder de cómputo elevado con un menor consumo energético, se puede considerar que son relativamente complicadas de programar, a pesar que para las tarjetas basadas en la arquitectura NVIDIA, cuentan con un superconjunto del lenguaje C, llamado CUDA C. Es por esta dificultad de programación que aparece un modelo basado en directivas como OpenACC, que permite indicarle al compilador cómo generar un programa paralelo en vez de que el desarrollador lo lleve a cabo a bajo nivel.

Finalmente, se demuestra que OpenACC es una herramienta que permite crear programas paralelos y acelerar su ejecución en tarjetas gráficas, comparable a la velocidad que una estación de trabajo de mayor costo puede proporcionar, sin llevar a cabo una programación de bajo nivel.

Agradecimientos

Mi más sincero agradecimiento a la Universidad Nacional Autónoma de México por haberme brindado la oportunidad de aprender de los mejores profesionistas.

Dr. Carlos Couder Castañeda, muchas gracias por su paciencia, conocimiento y experiencia porque sin ello este trabajo no hubiera sido posible.

A mis sinodales por su tiempo y sobre todo por compartir su conocimiento.

Agradezco a mi mami por todo su apoyo, por recordarme todos los días que soy capaz de cumplir mis objetivos y a mi papi que me observa desde el cielo, muchas gracias por siempre creer en mí.

A todos mis amigos que me ayudaron a crecer como persona: David Z, Yanahui, Berenice, Jovanny, Ramiro y Amalinalli.

A todos ustedes gracias por acompañarme en todo momento.

México, Pumas, Universidad, Goya! Goya!...

Índice general

Introducción 1
- Antecedentes 1
- Hipótesis 2
- Objetivo principal 3
- Organización del trabajo 3

1 GENERALIDADES DE LA PROGRAMACIÓN PARALELA 5
- 1.1 Antecedentes 5
- 1.2 Arquitecturas paralelas 6
  - 1.2.1 Una instrucción, un solo dato o SISD 6
  - 1.2.2 Múltiples instrucciones, un solo flujo de datos o MISD 7
  - 1.2.3 Una instrucción, múltiples datos o SIMD 7
  - 1.2.4 Múltiples instrucciones en múltiples datos o MIMD 7
- 1.3 Programación paralela 7
  - 1.3.1 Descomposición de programas 8
- 1.4 Modelos de programación paralela en Memoria Compartida 9
- 1.5 Programación en GPU 12
- 1.6 Dinámica de Fluidos Computacional 13

2 EL MODELO DE PROGAMACIÓN OPENMP Y OPENACC 17
- 2.1 Arquitectura de sistemas de memoria compartida 17
- 2.2 Arquitectura de los GPU 18
  - 2.2.1 Tarjetas NVIDIA 19
  - 2.2.2 Xeon Phi 21
  - 2.2.3 AMD 22
- 2.3 El modelo de programación de OpenMP 23
  - 2.3.1 Modelo de ejecución de OpenMP 23
  - 2.3.2 Modelo de memoria OpenMP 25
  - 2.3.3 Comandos de interés para el desarrollo de programas en OpenMP 26
- 2.4 El modelo de programación de OpenACC 28
  - 2.4.1 Modelo de ejecución de OpenACC 28
  - 2.4.2 Modelo de memoria de OpenACC 30

3 EL ESQUEMA NUMÉRICO DEL CASO DE ESTUDIO 33
- 3.1 Variables y parámetros 35
  - 3.1.1 Propiedades de los fluidos 36
  - 3.1.2 Tipo de fluidos 37
- 3.2 Ecuaciones que gobiernan el flujo 38
- 3.3 Transformación curvilínea 41
- 3.4 Condiciones a la frontera 43

3.5 Esquema numérico 44
4 DISEÑO E IMPLEMENTACIÓN DE LA APLICACIÓN EN OPENACC 49
4.1 Análisis del algoritmo 49
4.2 Vectorización de los ciclos 50
4.3 Diseño en OpenMP 53
4.4 Diseño en OpenACC 55
5 PRUEBAS DE RENDIMIENTO Y RESULTADOS DE LA SIMULACIÓN 59
5.1 Experimentos en CPU 59
5.2 Experimentos en la GPU 65
5.3 Validación del código traducido 65

BIBLIOGRAFÍA 75

Índice de figuras

Figura 1.1 Modelo Ian Foster para descomposición de programas 9
Figura 1.2 Metodología para la simulación de un CFD 14
Figura 2.3 Arquitectura de un sistema de memoria compartida 17
Figura 2.4 Estructura física de una tarjeta gráfica 19
Figura 2.5 Estructura física de un multiprocesador 20
Figura 2.6 Arquitectura APU (*Access Process Unit*) 22
Figura 2.7 Modelo de ejecución Fork-Join de OpenMP 24
Figura 2.8 Modelo de ejecución Fork-Join en regiones paralelas anidadas 24
Figura 2.9 Comportamiento de la memoria en una variable privada 25
Figura 2.10 Comportamiento de la memoria en variables compartidas 26
Figura 2.11 Modelo de ejecución OpenACC 29
Figura 2.12 Niveles de paralelismo en OpenACC 30
Figura 3.13 Movimiento infinitesimal 34
Figura 3.14 Ilustra la viscosidad lineal ν a través del canal y el esfuerzo de corte τ 37
Figura 3.15 Fuerza de presión 37
Figura 3.16 Plano computacional. 42
Figura 3.17 Plano computacional. 42
Figura 4.18 Diagrama de flujo del algoritmo numérico que emplea un esquema *predictor-corrector* 51
Figura 4.19 Diagrama de flujo del algoritmo numérico que emplea un esquema *predictor-corrector* 52
Figura 4.20 Diseño en OpenMP. 55
Figura 4.21 Modelo de aceleración en OpenACC 56
Figura 5.22 Comportamiento de afinidades en un SMP con dos Sockets y seis núcleos por socket, con HT desactivado; (a) asignación compacta, (b) asignación por dispersión. 61
Figura 5.23 Comportamiento de afinidades en un SMP con dos sockets y seis núcleos por socket, con HT habilitado; (a) afinidad compacta (b) afinidad dispersa. Los rectángulos significa que los núcleos virtuales son manejados por el mismo núcleo físico. 61
Figura 5.24 Tiempo de computo obtenido en un CPU Xeon Dual. 64

Figura 5.25 Comparación de los factores de velocidad contra la ley de Amdahl con una fracción serial de $f = 0{,}05$ y $f = 0{,}10$. En el inciso (a) se muestran un comportamiento gobernado por la ley de Amdahl, en el inciso (b), se muestra como la afinidad en un principio tiene una mayor rendimiento, esto es debido a que se utiliza un núcleo físico, pero cuando cuando se empieza a procesar dos hilos por núcleo físico, el rendimiento es muy similar. 64

Figura 5.26 Ubicación del visor numérico cerca de la zona de salida. El espesor de la zona de absorción CPML es de 10 cm ($10 \times \Delta x$). 67

Figura 5.27 Comparación de la energía total y el número de Mach correspondiente al visor ν. Simulaciones en GPU comparado con la solución serial de referencia 68

Figura 5.28 Gráficas de contorno del número de Mach (M) obtenido en diferentes tiempos de la simulación a 80 mil, 2 millones, 4 millones, 8 millones y a 10 millones de iteraciones de arriba hacia abajo. 70

Figura 5.29 Gráficas de contorno de la densidad (ρ) obtenido en diferentes tiempos de la simulación a 80 mil, 2 millones, 4 millones, 8 millones y a 10 millones de iteraciones de arriba hacia abajo 70

Figura 5.30 Gráficas de contorno de la temperatura (T) obtenida en diferentes tiempos de la simulación a 80 mil, 2 millones, 4 millones, 8 millones y a 10 millones de iteraciones de arriba hacia abajo. 71

Índice de tablas

Tabla 1 Resultados de tiempo de cómputo y *speed-up* obtenidos con OpenMP sobre Xeon-CPU con HT desactivado y afinidad compacta. Los factores de *speed-up* se calcularon con la mejor version serial optimizada como referencia. Con HT-Hyper Threading, V-VectorizaciÓn, D-Desactivado and A-Activado 63
Tabla 2 Tiempo de cómputo con su respectivo *speed-up* obtenido con HT habilitado para afinidad compacta y afinidad por dispersión. HT-Hyper Threading, V-Vectorización, A-Activo. 63
Tabla 3 Comparación del factor de velocidad tomando como referencia la GPU. 66
Tabla 4 Energía total y número de Mach obtenido en el visor V para las CPU y GPU al final de la simulación de 3 seg. 68
Tabla 5 El porcentaje de error relativo para la energía total y el número de Mach obtenido en el visor V del GPU comparado contra la versión serial en el CPU. 68

Lista de códigos

Código 1 Clausula REDUCTION 27
Código 2 Clausula PARALLEL 58
Código 3 Actualización de la memoria en el host 58

Definiciones y Acrónimos

OPENMP. Es una API que define un conjunto de directivas, funciones y variables de ambiente que permite al programador expresar paralelismo.

OPENACC. Es una API que sigue el modelo de directivas de OpenMP pero enfocado a la ejecución en tarjetas gráficas.

OPENCL. Es un lenguaje para la creación de aplicaciones paralelas basadas en tareas y datos. Su objetivo es la portabilidad por lo que se puede ejecutar en distintas plataformas, sistemas operativos y procesadores multicore.

MPI. Es un estándar para el paso de mensajería enfocado a la comunicación entre procesos distribuidos en múltiples procesadores. Los elementos principales de la biblioteca que intervienen en la comunicación es el un proceso que envía, el mensaje (datos) y el proceso que recibe.

CLÁUSULA. Es una palabra reservada integrada en las directivas para modificar la ejecución o la gestión de datos en un kernel.

DIRECTIVA. Es una palabra reservada especificada en el código por el programador, que indica cómo será ejecutado el código.

HYPER-HILADO, (HT). Es la capacidad que tiene el microprocesador para conmutar rápidamente dos hilos que usan el mismo núcleo

AFINIDAD. Se refiere a la manera (compacta o dispersa) en la que son asignados los hilos a cada núcleo.

FLUJO SUPERSÓNICO. Se define como un conjunto de moléculas dispersas con un espacio vacío entre ellas, que viajan a una velocidad superior a la velocidad del sonido.

VISOR NUMÉRICO. Son los puntos del dominio discreto donde se almacenan los valores de las variables primitivas para la evaluación de la transformación.

TECPLOT. Es un software de visualización de resultados especializado en dinámica de fluidos computacional CFD.

QTPLOT. Es un software de visualización de resultados de manera gráfica.

EYECTOR DE VACÍO. Es una bomba que incrementa la presión de un fluido con ayuda de otro llamado fluido motor inyectado a alta velocidad.

MULTIPROCESADOR. Forma parte del hardware y es aquel que dispone de microprocesadores para distribuir su carga de trabajo entre ellos.

CUDA CORE. Son núcleos más pequeños y optimizados enfocados al procesamiento de tareas de forma simultánea.

NÚCLEO DE PROCESAMIENTO. Forma parte de hardware pues se encuentra integrado en los multi-procesadores y son aquellos que ejecutan las instrucciones o tareas contenidas en los hilos de forma serial, además cuenta con una unidad ALU para realizar cálculos matemáticos.

WARP. Son bloques de 32 hilos en la tecnología CUDA.

AUTO-TUNING. Se le conoce como ajuste automático de rendimiento que genera combinaciones de rutinas y recursos por medio de búsqueda heurística.

SOCKET. Es la placa donde se encuentran los procesadores.

MEMORIA GLOBAL. - Es aquella a la que es accesible por todos los hilos de la aplicación en modo lectura-escritura al igual que el host.

MEMORIA COMPARTIDA. Es una memoria de acceso rápido común a todos los hilos que componen un bloque (block).

DOMINIO. Es el área de estudio limitada por criterios físicos o lógicos.

GRID. Es la malla que divide discretamente al dominio para resolver la simulación con ayuda de métodos numéricos.

BLOCK. Es el conjunto finito de hilos que se ejecutan concurrentemente en un multiprocesador.

SPEED-UP. Es un índice que nos ayuda a identificar si aumentó el rendimiento en una aplicación ejecutada con p procesadores con respecto a la versión serial. El speed-up toma los valores de cero a `p` donde lo ideal es que sea un valor cercano a `p`.

LEY DE AMDHAL. Es un parámetro que mide la ganancia del rendimiento obtenido a partir del porcentaje de código secuencial y la porción paralelizable.

COMPUTADORA PARALELA. Es un conjunto de procesadores que trabajan simultáneamente conectados por medio de una red para resolver problemas computacionales de manera rápida.

REGIÓN PARALELA. Es aquel código que será ejecutado de manera paralela en el dispositivo acelerador mientras se mantienen las variables en memoria.

.

CLUSTER. Es un tipo de computadora paralela donde la conexión entre los procesadores se realiza entre computadoras.

BANDERAS DE COMPILACIÓN. Son instrucciones que se especifican en la compilación y modifican el modo en que se crea el archivo binario.

HILO. Es la unidad mínima de procesamiento que puede gestionar el sistema operativo. Comparten memoria, recursos y datos de manera que el manejo de los hilos se vuelve complicado al mantener la coherencia de datos, por lo que se recurren a secciones críticas.

PROCESO. Es cualquier programa en ejecución por lo que se compone de instrucciones, se asigna memoria y un estado. Estos recursos no se comparten entre ellos, sin embargo la comunicación puede existir mediante sockets.

KERNEL. Es un conjunto de instrucciones que se ejecutarán en una tarjeta gráfica por N número de hilos.

HT Hyper-Threading. Hyper-Hilado

API Application Programming Interface. Interfaz de Programación de Aplicaciones

CUDA Compute Unified Device Architecture. Arquitectura Unificada de Dispositivos de Cómputo

PGI Portland Group Incorporation

PML Perfectly Match Layer. Capa perfectamente ajustada

HPC High performance computing. Computo de alto rendimiento

Introducción

Antecedentes

La simulación numérica actualmente se puede considerar como una herramienta que ha permitido la simulación de fenómenos físicos donde la experimentación no es posible o simplemente es muy costosa. No obstante, los requerimientos de la simulación numérica demandan de un mayor poder de cómputo que trajo consigo la creación de la computación paralela, que permite la simulación de fenómenos con una mayor precisión y un mayor número de parámetros.

En los últimos cinco años han emergido sistemas orientados al procesamiento numérico basado en arquitecturas paralelas con una vertiginosa actualización. Prácticamente cada seis meses emerge una nueva tarjeta NVIDIA o cada año un coprocesador Xeon Phi. Sin embargo, la implementación de un modelo en una computadora paralela es un trabajo arduo, debido a que se requieren modificar los algoritmos computacionales para que se puedan ejecutar en un entorno de multiprocesamiento y es en este contexto donde se requieren metodologías y técnicas de mayor abstracción, que minimicen el costo de implementación y en este sentido hacen su aparición OpenMP y OpenACC, el primero orientado a arquitecturas multi-núcleo de memoria compartida y el segundo a unidades de procesamiento gráfico (GPUs).

Las arquitecturas de memoria compartida son cada vez más comunes en el mercado de los ordenadores de alto rendimiento. Los avances en la tecnología del hardware nos permiten tener varios núcleos de procesamiento con su respectiva unidad de punto flotante, compartiendo el mismo banco de memoria. Para facilitar la programación multi-núcleo sobre estas arquitecturas es necesario utilizar un esquema de programación como OpenMP el cual ha mostrado ser lo suficiente eficiente para la implementación de algoritmos numéricos en máquinas multi-núcleo de memoria compartida.

OpenMP está sustentando en una combinación de funciones y/o directivas y es sumamente portable sobre todo en máquinas con sistemas operativos Linux, FreeBSD e incluso Windows, además permite que la tarea de cómputo se pueda dividir en granularidad intermedia o fina. Para lograr el uso de los múltiples núcleos crea hilos de ejecución, cada uno de estos es atendido por un núcleo y cada hilo puede acceder a la misma memoria global y tener su propia memoria privada, por lo anteriormente expuesto se puede decir que OpenMP ya es un estándar

en la programación de los sistemas de altas prestaciones.

No obstante, con la aparición de la tecnología CUDA, basada en el uso de tarjetas gráficas como procesadores numéricos, se ha vuelto indispensable traducir o convertir los códigos a este nuevo paradigma, ya que promete un mejor rendimiento y una considerable reducción del tiempo de cómputo, bajando con ello los costos de adquisición del hardware y de consumo energético. Sin embargo, la programación en CUDA está basada la creación de los kernels (funciones concurrentes), y no siempre se tiene claro el diseño de estos, por lo que puede convertirse en una tarea tediosa y complicada, lo que significa una inversión adicional de tiempo de programación y que pueda ser más propensa a errores.

De la necesidad de programar con mayor facilidad las tarjetas gráficas NVIDIA, nace OpenACC como una metodología de programación análoga a la utilizada en OpenMP basada en directivas. OpenACC está dirigido a los desarrolladores de aplicaciones que requieren beneficiarse de aceleradores muti-núcleo como las unidades de procesamiento gráfico de NVIDIA con un modelo de programación más simple, con la idea que no solo el código generado sea exclusivo para tarjetas NVIDIA, sino también para algunas otras tarjetas aceleradoras como ATI.

Hipótesis

La hipótesis central de este trabajo se basa en que si utilizamos una tarjeta gráfica y se programa utilizando OpenACC y lo aplicamos a un código numérico se debe obtener un rendimiento mejor o similar a un menor costo de hardware y energía que que utilizando OpenMP en una máquina multinúcleo.

Los artículos de investigación consultados en el estado del arte muestran que el rendimiento que se puede obtener utilizando una tarjeta gráfica y OpenACC puede ser similar o mejor que el de una máquina multinúcleo, sin necesidad de hacer una programación a bajo nivel, no obstante, indican que si no se realiza un diseño adecuado de la adaptación del código fuente puede suceder una baja de rendimiento en vez de una mejora (Reyes et al., 2013; Reyes, López-Rodríguez, Fumero and De Sande, 2012a; Reyes, López, Fumero and De Sande, 2012a; Bednarz et al., 2011; Reyes, López, Fumero and De Sande, 2012b; Sabne et al., 2012; Du et al., 2012; Wienke et al., 2012a; Beyer et al., 2011; Amritkar et al., 2012).

Para demostrar cómo se tiene que hacer un buen diseño de adaptación, se eligió un código para la simulación de flujo supersónico en eyectores basado en diferencias finitas (Couder-Castaneda, 2009; Martin and Couder-Castaneda, 2010a), y se

pretende reducir el tiempo de cómputo significando una menor espera en el cálculo de las simulaciones que se traduce en un ahorro económico.

Objetivo principal

Adaptar un código numérico basado en diferencias finitas para simular flujo supersónico en eyectores escrito utilizando OpenMP en FORTRAN al modelo de aceleración de OpenACC, con la finalidad de reducir el tiempo de cómputo minimizando los costos del hardware. Mostrando que se requiere hacer un buen diseño para lograr dicho propósito.

Los objetivos particulares son los siguientes:

- Llevar a cabo un estudio general sobre el paradigma de la programación paralela.
- Hacer una investigación sobre las instrucciones y metodologías sobre el uso de OpenMP y OpenACC
- Analizar el algoritmo del código que se pretende migrar a de OpenMP a OpenACC.
- Realizar una estrategia de diseño en OpenACC
- Llevar a cabo la codificación e implementación en OpenACC.
- Realizar todas las pruebas de rendimiento en multi-núcleo y GPU.

Organización del trabajo

La tesis se organiza como sigue:

En el capítulo 1, se da un panorama general de la programación paralela y sus diferentes paradigmas, para contextualiza el marco teórico del las metodologías utilizadas en este trabajo.

En el capítulo 2, se abordan las particularidades de los paradigmas OpenMP y OpenACC y las arquitecturas en las cuales operan.

En el capítulo 3, se describe el modelo numérico que servirá como caso de estudio para las pruebas de rendimiento. Este modelo ya ha sido aplicado dentro de la industria petrolera, para el flujo supersónico en eyectores y es de primordial interés reducir su tiempo de cómputo.

.

En el capítulo 4 se aborda el diseño propuesto de cómo debe ser una aplicación OpenACC a partir de OpenMP.

En el capítulo 5 se presentan las pruebas de rendimiento de las implementaciones en OpenMP y OpenACC, así como los resultados de una simulación numérica.

Finalmente se dan las conclusiones y las referencias a fuentes consultadas.

Capítulo 1

Generalidades de la programación paralela

En este capítulo se presenta un panorama general de la programación paralela, sus diferentes paradigmas y algunos trabajos relacionados con códigos de dinámica de fluidos. También se exponen algunos estudios de rendimiento entre las diferentes plataformas de programación paralela para contextualizar el marco teórico de las metodologías utilizadas en este trabajo.

1.1 Antecedentes

Hasta hace unos años el avance en la arquitectura de las computadoras estaba enfocado en la evolución del hardware. Se pensaba que el rendimiento sólo se podía aumentar creando procesadores más rápidos y eficientes que eran por supuesto muy costosos. Pero a medida que los desarrolladores de hardware progresaron e instrumentaron múltiples núcleos en un único chip, se dio pie al procesamiento en paralelo, de manera que se podían realizar múltiples tareas al mismo tiempo.

El mundo de la computación de alto rendimiento o HPC (High Performance Computing) por sus siglas en inglés, está experimentando cambios rápidos y se ve reflejado en arquitecturas informáticas como las unidades de procesamiento gráfico o GPU, capaces de aumentar el rendimiento a un bajo costo pero a una alta capacidad de cómputo.

Las personas asocian las GPU con la presentación de imágenes a gran velocidad y no se equivocan. Sin embargo, los científicos aprovecharon su velocidad en procesos matemáticos implementando unidades de punto flotante, lo que permitió mejores cálculos dio más realismo a los gráficos.

Hasta antes del 2003 la principal aplicación de estos dispositivos aceleradores fue dirigida a la industria de los videojuegos. El primer intento de cálculo no gráfico en las GPU fue una multiplicación de matrices. Una vez implementado el cálculo con unidades de punto flotante se puso a prueba con una factorización LU convirtiéndose en uno de los primeros kernels optimizados. (Du et al., 2012).

La programación paralela ha revolucionado en gran medida a las disciplinas científicas y se ha aplicado al procesamiento de imágenes, simulación de partículas, modelos oceánicos, extracción segura de hidrocarburos, aerodinámica y por supuesto en dinámica de fluidos. Por ejemplo, una aplicación en el campo de la medicina es la magneto encefalografía, donde se mide el campo magnético inducido por la densidad de corriente en el interior del cerebro humano fuera de la cabeza. Para reconstruir la actividad focal en el cerebro es necesario resolver el problema inverso neuromagnético, que no es más que un problema de optimización no lineal, compuesto de una función objetivo dedicada a encontrar el valor mínimo de "p-norm" y derivadas de primer y segundo orden para asegurar la convergencia computacional. La tarea de la programación paralela es resolver el producto matriz-vector de dimensiones 128 x 512000 en el menor tiempo posible (Wienke et al., 2012b).

La computación desde sus inicios se ha considerado como una herramienta para solucionar problemas que ayuden al ser humano a comprender los fenómenos físicos que gobiernan nuestro entorno y ayudar en gran medida a nuestro bienestar. La simulación computacional permite estudiar los fenómenos de forma segura y tiene como objetivo disminuir el costo en tiempo y dinero, que probablemente serían cantidades exageradas si se experimentara en la realidad.

1.2 Arquitecturas paralelas

Una computadora paralela es un conjunto de procesadores que trabajan simultáneamente conectados por medio de una red para resolver problemas computacionales de manera rápida. Existen dos tipos de conexiones entre estos procesadores y se definen por su ubicación. Si dichas conexiones se encuentran dentro de la misma computadora, se le llama sistema de programación paralelo. De igual manera, si la conexión se realiza entre computadoras, se le conoce como sistema de programación distribuida o *cluster* donde la configuración de memoria también es distribuida y cada procesador tiene su propia memoria.

Una de las clasificaciones de las arquitecturas paralelas más utilizadas es la taxonomía de Flynn's (Snyder, 1988), su clasificación distingue las arquitecturas de acuerdo con las dimensiones del conjunto de instrucciones secuenciales que se ejecutan en un procesador y el flujo de datos como se enuncia a continuación.

1.2.1 *Una instrucción, un solo dato o SISD*

Utiliza el concepto de arquitectura de Von Neumann donde, se asume que un procesador ejecuta sólo una única instrucción a la vez, por ejemplo: la dirección

de un dato que se lee o se escribe en la memoria, una operación aritmética o la dirección de la siguiente instrucción a ejecutar. Todas las máquinas SISD aseguran la ejecución en serie del programa, por medio de un registro simple llamado contador de programas, es por eso que se les conoce como computadoras seriales. La función del contador es apuntar a la siguiente instrucción a procesar.

1.2.2 *Múltiples instrucciones, un solo flujo de datos o MISD*

Esta arquitectura consiste en aplicar varias instrucciones a un único flujo de datos. Un ejemplo de lo anterior es un mensaje cifrado definido por un conjunto de datos finitos, al que se le aplicamos diferentes operaciones que forman parte de un algoritmo de descifrado para intentar conocer el mensaje. Cabe señalar que existen muy pocos ejemplos reales de esta arquitectura.

1.2.3 *Una instrucción, múltiples datos o SIMD*

Esto significa que una única instrucción se aplica sobre diferentes datos al mismo tiempo. El único inconveniente que presenta esta arquitectura es que para ejecutar otra instrucción es necesario que esta última se haya realizado en todos los datos, por lo que si se desea hacer una operación elemental por renglón donde, se multiplica una constante por todo el vector y después se suma a otro vector, es necesario que la multiplicación se realice en todos los elementos para después aplicar la suma.

1.2.4 *Múltiples instrucciones en múltiples datos o MIMD*

Es la arquitectura más compleja, pues soluciona la limitante que presenta la arquitectura SIMD. En esta arquitectura es posible ejecutar múltiples instrucciones sobre diferente grupo de datos en forma simultánea, ya sea en un cluster o multiprocesador donde cada procesador ejecuta su propio flujo de instrucciones sobre su propio flujo de datos, permitiendo la ejecución concurrente o paralela.

1.3 Programación paralela

La programación paralela es la ejecución simultánea de recursos computacionales con el fin de resolver problemas en el menor tiempo posible y su principal reto para lograr ese propósito es dividir los problemas grandes en componentes que puedan ser ejecutados en forma concurrente.

1.3.1 *Descomposición de programas*

Existen tres tipos de descomposición. La *descomposición trivial* es la más simple, pues no implica paralelizar el código, sino que este se ejecuta en varios procesadores paralelamente, cada uno con sus propios parámetros de entrada. Para poder aplicar este método es necesario que cada parte del código sea independiente una de la otra porque no hay comunicación entre procesadores. En la *descomposición funcional* el trabajo se divide en tareas que se llevan a cabo en diferentes procesadores. La diferencia con la descomposición trivial es que estos procesadores se comunican entre sí; cuando un procesador termina una tarea se la pasa al otro procesador, es decir, opera de la misma manera que una línea de producción de una fábrica. Por último la *descomposición de dominio* asigna un conjunto de datos a cada procesador de manera que tenga la misma carga de trabajo; dicha sincronización de datos es responsabilidad del programador.

Foster's (1995) propone explotar las características del algoritmo para hacer un buen diseño paralelo independientemente de las especificaciones de la máquina y propone dividir el proceso en cuatro etapas: partición, comunicación, aglomeración y mapeo.

PARTICIÓN. Como su nombre lo indica, consiste en identificar las partes del código que en principio sean independientes, para poder agruparse en tareas y posteriormente ejecutarse en paralelo. Se aconseja dividir en múltiplos del número de procesadores con los que se cuente, sin embargo en una buena partición se empieza por dividir el cálculo asociado al problema (descomposición de dominio) y posteriormente los datos sobre los que opera (descomposición funcional).

COMUNICACIÓN. Las tareas generadas por una partición están destinadas a ejecutarse al mismo tiempo pero no a ejecutarse en forma independiente, pues un cálculo relacionado con una tarea podría necesitar datos asociados a otra tarea; este flujo de información se especifica en esta fase. Es aquí donde se busca optimizar el rendimiento mediante la distribución y organización de acciones de comunicación sobre las tareas de manera que permita la ejecución paralela.
Existen ocho tipos de comunicación: local, global, estructurado, no estructurado, estática, no estática, sincronizada y no sincronizada.

AGLOMERACIÓN. Esta etapa está enfocada a mejorar el rendimiento o reducir el costo de desarrollo aumentando la granularidad, de ser necesario las tareas serán reagrupadas y la comunicación entre ellas se verá modificada. Sin embargo, se puede modificar el rendimiento al reducir la cantidad de tiempo dedicado a la comunicación. Claramente esta mejora se puede lograr mediante el envío de menos datos o reduciendo el número de mensajes, incluso si se envían muchos

.

datos. Esto ocurre debido a que cada comunicación no sólo incurre en un costo proporcional a la cantidad de datos trasferidos, sino también un costo por trasferencia.

MAPEO. Se especifica dónde se ejecuta cada tarea y pueden ocurrir dos situaciones: Asignar tareas que son capaces de ejecutarse simultáneamente en diferentes procesadores con el fin de aumentar el paralelismo o asignar tareas que se comuniquen con frecuencia en el mismo procesador aumentando la localidad.

Lo anterior se puede resumir que las dos primeras etapas se enfocan en el paralelismo y la escalabilidad. La búsqueda de algoritmos con estas características tiene como finalidad poder importarlo a otras plataformas sin tener que planificar desde cero. Las últimas dos etapas se centran en la localidad de los datos y el rendimiento. En la Figura 1.1 se muestra el modelo de Ian Foster.

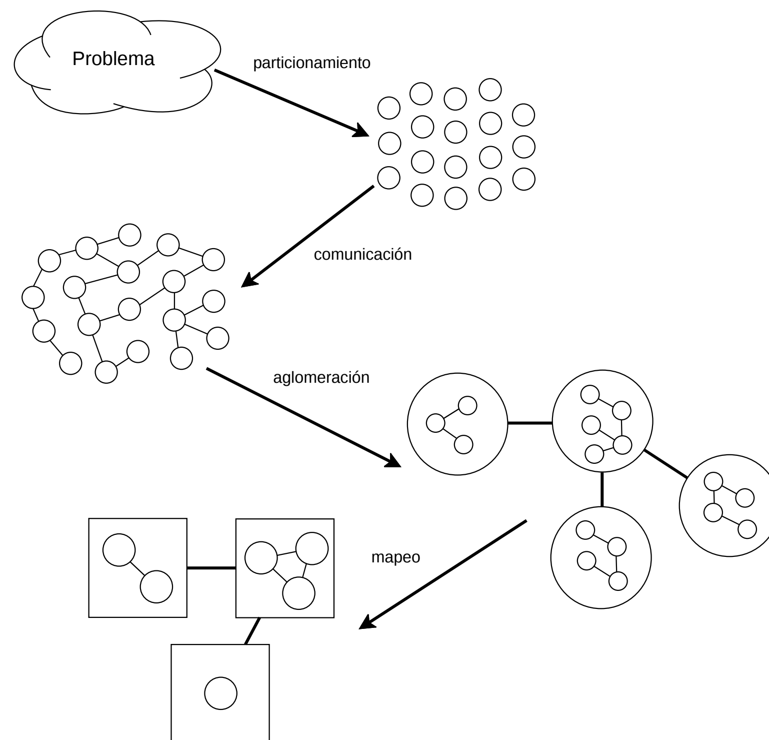


Figura 1.1: Modelo Ian Foster para descomposición de programas

1.4 Modelos de programación paralela en Memoria Compartida

Los modelos de programación propuestos para los procesadores multi-core o multi-núcleo se pueden clasificar de acuerdo con el modelo de comunicación uti-

lizado.

Los modelos de programación paralela basados en memoria compartida se comunican compartiendo los datos en la memoria global, esto quiere decir que cualquier dirección de memoria es accesible desde cualquier núcleo, por lo que las direcciones son únicas. La principal desventaja es mantener la consistencia de datos cuando diferentes procesadores se comunican y comparten el mismo elemento de datos haciendo uso de los protocolos de coherencia de cache.

En modelos de programación basados en memoria distribuida la comunicación entre procesadores se realiza mediante envío explícito de mensajes. Los mensajes viajan por medio de una red de interconexión y los datos e instrucciones se almacenan en la memoria local de cada procesador. Es posible ejecutar las mismas instrucciones para diferentes datos, incluso diferentes instrucciones en cada procesador; esto permite tener diferentes tipos de paralelismo en un mismo código.

El presente trabajo está enfocado a los modelos de programación paralela basada en memoria compartida. Saber el comportamiento, manejo de datos, memoria y arquitectura física en la cual se desempeñan es esencial para obtener el mayor beneficio al adaptar algoritmos a aplicaciones paralelas. Las arquitecturas de memoria actualmente más comunes son los GPUs y las arquitecturas multi-núcleo.

Para llevar a cabo la programación en GPU, se han propuesto muchos modelos de programación de alto nivel, entre los cuales podemos destacar a OpenCL, que es un lenguaje para la creación de aplicaciones paralelas basadas en tareas y datos. Funciona en GPUs, sistemas operativos y procesadores multicore y su objetivo principal es proporcionar portabilidad, es decir, permitir que una sola aplicación OpenCL funcione en diferentes plataformas de hardware. Sin embargo en el trabajo de (Du et al., 2012) los experimentos mostraron que aunque los kernels se podían ejecutar en cualquier plataforma el rendimiento no era el mismo y por lo tanto el rendimiento no era portable. Para hacer frente a esta desventaja emplearon una técnica llamada auto-tuning o ajuste automático de rendimiento que genera combinaciones de rutinas y recursos por medio de búsqueda heurística. El resultado es el mejor espacio de parámetros para dichos kernels. Con este procedimiento aseguran la portabilidad del rendimiento sólo sí se aplica cada que se ejecute en una nueva plataforma.

Para los GPUs de la marca NVIDIA se desarollo CUDA, el cual es un conjunto de instrucciones adicionales a los lenguajes FORTRAN y C, las construcciones son similares entre si pero CUDA FORTRAN tiene directivas especiales que pueden generar automáticamente los kernels para casos comunes. CUDA organiza hilos en bloques para después ejecutar el mismo programa en todas las unidades de

.

procesamiento, de modo que sigue un modelo de programación SPMD. Los bloques se dividen automáticamente en *warps* y se ejecutan como SIMD.

La obtención de rendimiento en códigos CUDA y OpenCL es difícil en si misma, ya que requiere una comprensión profunda de la arquitectura subyacente. Por ejemplo, OpenCL maneja el concepto de memoria compartida como memoria local con la sentencia `LOCAL-DOUBLE`, además hace diferencia explicita entre las direcciones de memoria global (espacio de direcciones en la memoria del dispositivo) y las direcciones de memoria local (variable de registro o puntero a la memoria compartida).

Aunque CUDA, OpenCL sean entornos maduros y existe documentación que permite a los usuarios hacer aplicaciones para GPU, existe la desventaja de reescribir el código para lograr un buen factor de rendimiento, es por ello que se ha investigado muchos modelos de programación basados en directivas con el objetivo de obtener mayor rendimiento a menos costo de programación.

Para la programación en arquitecturas multi-núcleo tenemos OpenMP que es una interfaz de programación de aplicaciones en memoria compartida, el cual consiste en una conjunto de directivas, una librería de funciones y variables de entorno. OpenMP adopta el modelo fork-join, el cual consiste en un hilo principal que empieza la ejecución en forma secuencial hasta encontrarse con un constructor paralelo donde se crea un grupo de hilos. Dentro de la región paralela cada hilo puede tener sus propias variables privadas y trabaja con su propio segmento de datos. Cuando surge la necesidad de comunicación entre hilos es necesario hacer uso de las variables compartidas.

OpenACC comparte el enfoque de OpenMP, sólo que dirigido a GPUs, donde el programador anota el código secuencial combinado con las directivas del compilador e identificar las áreas de código que pueden ejecutarse en la tarjeta gráfica. Con la información proporcionada por las directivas, el compilador es capaz de gestionar el movimiento de datos entre el acelerador y la memoria del host así como el almacenamiento de los mismos en tiempo de ejecución. Las directivas están disponibles tanto en C como en FORTRAN y para ambos el compilador genera un archivo binario único que contiene las versiones GPU y CPU del código, por lo tanto, este binario puede ser ejecutado en plataformas que no cuenten con GPU. Las especificaciones del modelo de memoria y ejecución de OpenMP y OpenACC se profundizan en el siguiente capítulo.

Muchos desarrolladores han definido sus propias extensiones al lenguaje C para tratar el paralelismo en plataformas personalizadas orientadas a compiladores código a código. Por ejemplo, OpenMPC es una plataforma de programación que

.

genera códigos CUDA de códigos OpenMP, consiste en una interfaz de programación OpenMP extendida, un traductor fuente a fuente y un sistema de modificación automática. El compilador identifica el segmento de código que se puede optimizar y sustituye el código con sentencias CUDA de acuerdo a la mejor combinación de optimizaciones. Las opciones de optimización consisten en la configuración de entorno del programa, estrategias de almacenamiento de datos en memoria cache, descarga de datos, traducción del código, entre otras (Lee et al., 2009).

Por otro lado HiCUDA proporciona al desarrollador un conjunto de directivas que se asignan a las operaciones habituales de CUDA. El dispositivo de HiCUDA analiza el código original usando la interfaz GNU-3, después opera sobre Open64 para reemplazar las directivas. Primero extrae los ciclos y después reemplaza las llamadas a CUDA en tiempo de ejecución. Los kernels son automáticamente extraídos desde el archivo fuente original; una vez identificados, la distribución de las iteraciones entre los hilos y bloques se hace de acuerdo a lo que especifique la clausula `LOOP` (Han and Abdelrahman, 2011).

Por último, la universidad de La Laguna en España desarrolló una aplicación de OpenACC llamada AccULL. Se compone de un compilador fuente a fuente (YaCF) perteneciente a previas investigaciones y una biblioteca en tiempo de ejecución desarrollada desde cero, similar a un compilador pero en lugar de generar un archivo binario la aplicación genera una estructura jerárquica con instrucciones de compilación. El compilador posee módulos de Python que extrae el código del kernel y lo reemplaza con directivas CUDA u OpenCL haciendo llamadas a una biblioteca. La aportación principal de esta aplicación es el manejo de la existencia de varias instancias de una variable puesto que hoy en día los dispositivos aceleradores no comparten el espacio de direcciones del host. Para hacer frente a esta situación, la biblioteca utiliza un mecanismo de detección de direcciones mediante un puntero que coincida con cada variable del sistema principal y su contraparte en el dispositivo (Reyes, López-Rodríguez, Fumero and de Sande, 2012b).

1.5 Programación en GPU

La estructura de repetición que genera más carga computacional son los ciclos `FOR` en C o `DO` en FORTRAN, por lo que se busca aumentar el rendimiento de estas estructuras en las diferentes plataformas (Reyes et al., 2013). Cuando se ocupa el compilador de PGI la optimización de un loop requiere que cada iteración sea completamente independiente al resto. Por lo tanto, si el compilador no es capaz de garantizar esta situación no genera el kernel. Pero existe una forma de forzarlo usando la cláusula `intependent`, la única condición es que los índices sean decla-

rados privados.

Es recomendable usar la bandera de compilación `-Minfo` para mostrar información detallada sobre la generación de código CUDA debido a que en ocasiones el compilador no crea el kernel por una falsa independencia entre iteraciones, pero la compilación termina correctamente. El uso de banderas de compilación ayuda en gran medida a generar código más eficiente y aumenta el rendimiento de sus aplicaciones, sin embargo la eficiencia depende de la combinación de las banderas, de las características del programa (Sabne et al., 2012).

Es interesante el rendimiento que se puede lograr si se tiene conocimiento del modelo de memoria sin atarse a alguna plataforma en específico. El almacenamiento de los datos en CUDA ocurre de manera consecutiva a manera de vector unidimensional generando un costo de acceso. Imaginemos que se almacena una matriz cuadrada A de orden 3 y queremos acceder al elemento A[2,2]; realmente se accede a la posición 5 en un arreglo lineal. En la memoria cache no sucede eso, es decir, se respeta la posición original de la matriz convirtiéndose en una ventaja para un problema de diferencias finitas donde optimiza el acceso a los elementos vecinos, el inconveniente al usar memoria cache es el almacenamiento limitado.

La principal motivación para la comparación de rendimiento sobre los lenguajes para GPU como OpenACC y CUDA FORTRAN es la existencia de códigos escritos en FORTRAN nativo orientados a CPU, pues se ha demostrado que el compilador de FORTRAN suele generar códigos de computación científica más eficiente que los compiladores de C, además la traducción a estos entornos suele ser fácil e intuitiva excepto la traducción a CUDA C ya que es propenso a errores y requiere reescribir el código.

Los resultados dependen en gran parte de la complejidad del algoritmo, por ejemplo, la versión OpenACC para la ecuación de Navier-Stokes dio resultados similares con respecto a las otras versiones CUDA FORTRAN y C que fueron las más rápidas. Pero si tomamos en cuenta el esfuerzo para traducir códigos a CUDA nativo, por llamarlo de alguna manera, realmente no vale la pena sacrificar tiempo de programación por ganar 10 % más de rendimiento (Cloutier et al., n.d.).

1.6 Dinámica de Fluidos Computacional

Una de las disciplinas que más se ha beneficiado de la introducción de los GPUs es la dinámica de fluidos computacional (CFD). El objetivo principal de la simulación CFD es reconstruir la realidad del movimiento y comportamiento de los

fluidos sin los peligros y costos de la experimentación tradicional.

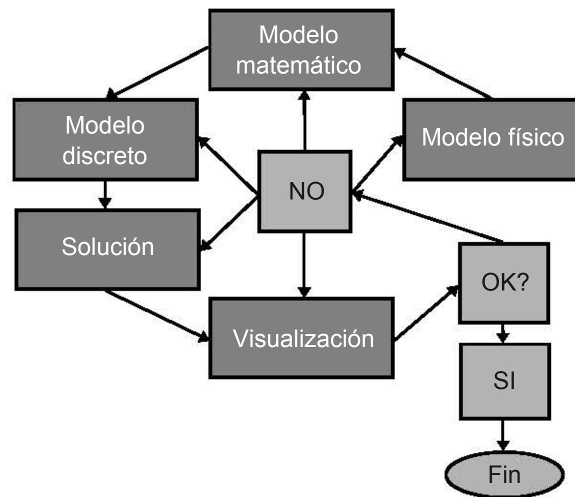


Figura 1.2: Metodología para la simulación de un CFD

La metodología contempla cinco etapas que interactúan entre ellas (ver Figura 1.2). El procedimiento consiste en aplicar los principios y leyes que gobiernan la naturaleza del fenómeno.

Una vez establecidas las condiciones físicas del modelo se describe el fenómeno mediante el uso de relaciones matemáticas como las ecuaciones de Navier-Stokes y las ecuaciones de energía para simular el movimiento de un fluido. En la etapa de pre-procesamiento se divide el dominio en un espacio discreto con el fin de resolver estas ecuaciones con algún método numérico.

El modelo que se aborda en este trabajo contempla una serie de variables y constantes, las leyes de conservación con la gravedad, el ángulo de la cámara de mezcla, la longitud de la garganta, la posición de la salida de la boquilla y los efectos de la temperatura y presión en el fluido para un modelo de gas ideal y gas real en un eyector de aire supersónico, que en conjunto proporcionan el realismo a la simulación.

La distribución de datos es un paso previo al desarrollo de algoritmos sofisticados, eficientes, seriales y paralelos que ayudaran a resolver el modelo. El método implementado en las investigaciones que forman parte del estado del arte consiste en identificar las partes del código que generan cálculo masivo para después clasificarlo en el nivel de paralelismo. Un ejemplo claro es el uso de OpenMP para desarrollar códigos de fluidos donde se implementan las celdas fantasma en la topología de bloques la cual tiene varias ventajas y la principal de ellas es la

.

portabilidad entre OpenMP, MPI e híbrida.

La visualización de los resultados es muy importante para el análisis e interpretación de los mismos. Tecplot y Qtplot son algunas de las herramientas que pueden ayudan a visualizar dichos resultados.

.

Capítulo 2

El Modelo de Progamación OPENMP Y OPENACC

Las arquitecturas de memoria compartida cada día predominan más en la industria del cómputo de alto rendimiento. Los avances en la tecnología del hardware han permitido el incremento del número de núcleos que comparten el mismo espacio de direcciones de memoria; como es el caso de las arquitecturas multiprocesador/multi-núcleo y las GPUs.

Actualmente es preferible que cada uno de los nodos de un sistema multicomputador comúnmente conocido como *cluster* tenga el mayor número de núcleos integrados en los procesadores o las GPUs que lo componen, con la finalidad de minimizar el costo de comunicación producido por el uso de la red en un modelo de paso de mensajes. En el presente capítulo se explicarán las ventajas de esta arquitectura.

2.1 Arquitectura de sistemas de memoria compartida

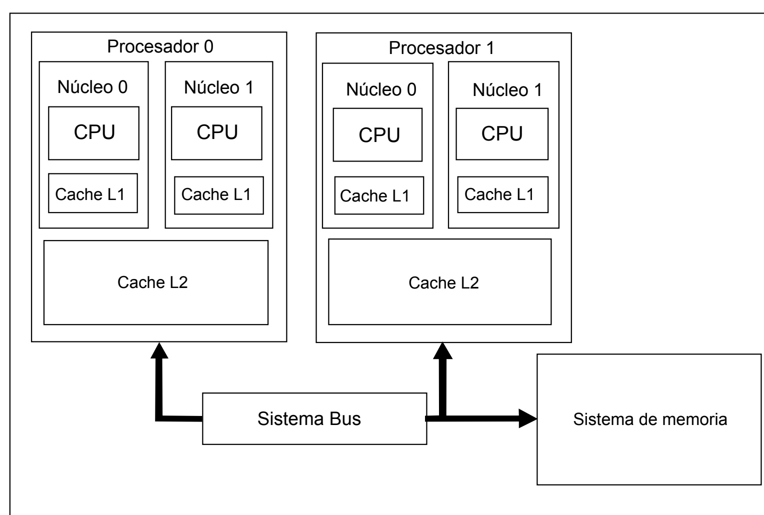


Figura 2.3: Arquitectura de un sistema de memoria compartida

Los multiprocesadores se agrupan de 2 en 2 o de 3 en 3

En una arquitectura de memoria compartida (ver Figura 2.3) cada procesador contiene varias unidades de proceso denominados núcleos que ejecutan hilos en modo SIMD, es decir cada hilo ejecuta una instrucción de forma simultánea simu-

lando un paralelismo perfecto. El desarrollo de procesadores multinúcleo impulsó el paralelismo basado en hilos, a la vez que las mejoras a nivel de hardware fueron significativas ya que producen poco calor y consumen menos energía.

Los hilos que trabajan en arquitecturas de memoria compartida pueden acceder a niveles internos de memoria cache pero también tienen acceso a la memoria que comparten los procesadores. En la memoria común se guardan los datos y el código que ejecutan los procesadores. La ejecución de instrucciones como la operación con datos, lectura y escritura en memoria se realizan por todos los procesadores en una misma señal de reloj es decir de forma síncrona.

El proceso de comunicación entre procesadores consiste en modificar las variables ubicadas en la memoria, pero si un hilo actualiza una ubicación de memoria y otro lee la misma dirección, o dos hilos almacenan un valor en la misma ubicación el hardware no garantiza la consistencia de los resultados.

2.2 Arquitectura de los GPU

La arquitectura de los GPU se compone de numerosos multiprocesadores paralelos. Internamente cada multiprocesador cuenta con un conjunto definido de núcleos, una unidad de funciones matemáticas (ALU) que puede realizar operaciones como raíz cuadrada o divisiones que, aunque son operaciones costosas no afectan el rendimiento porque se usan con poca frecuencia; también posee una memoria local de baja latencia unido a cada multiprocesador. La unidad de búsqueda y lanzamiento de instrucciones es el responsable de que las instrucciones se ejecuten en paralelo en todos los procesadores.

Un aspecto importante con respecto a la arquitectura física es la organización de la memoria, se sabe que la memoria de las GPU está organizada de manera jerárquica de acuerdo con el tamaño, ya que mientras más grande sea una memoria, es mayor el tiempo de acceso y como el objetivo es acelerar el intercambio de datos entre las unidades de proceso y la memoria del dispositivo se recurre al uso de los registros, memoria cache y memoria central (ver Figura 2.4).

La memoria cache está dividida en tres niveles y por cada nivel la latencia aumenta. Por ejemplo, si una instrucción de datos se almacena en la cache L1 (32 Kb) el resultado estará listo en cuatro o cinco ciclos de reloj; en la cache L2 (256 Kb) el resultado tomará 12 ciclos de reloj y 45 si se almacena en la memoria L3 (20 Mb). La cache es una memoria pequeña y de rápido acceso que accede entre cinco y diez veces más rápido que la memoria principal.

.

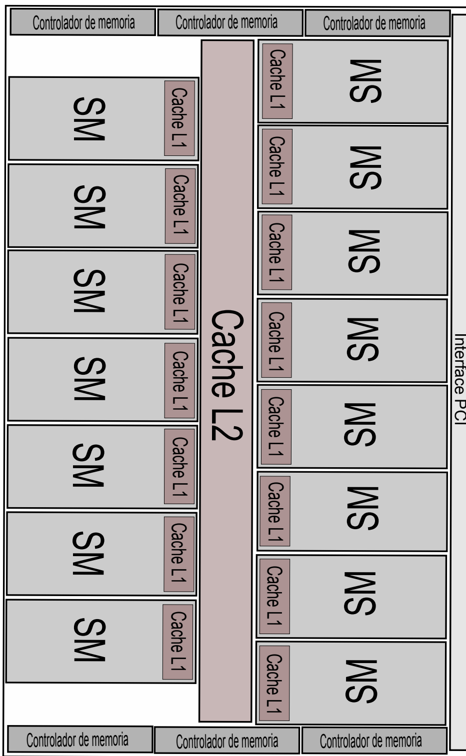


Figura 2.4: Estructura física de una tarjeta gráfica

La GPU está diseñada para acceder a los datos de forma estructurada, es decir, que los hilos acceden eficientemente a bloques contiguos de memoria. Los programas que no siguen este modelo se ejecutarán notablemente más lento porque hacen recuperaciones de memoria aleatoria o por acceder simultáneamente al mismo banco de memoria. Pero si el programa está estructurado para utilizar los datos contiguos, la memoria puede correr a todo ancho de banda.

2.2.1 *Tarjetas NVIDIA*

Las características físicas implementadas en la tarjeta Kepler DK210 de NVIDIA puesta al publico en el 2014 están enfocadas a explotar los 2,880 núcleos distribuidos en los 15 multiprocesadores (SMX). Cada SMX tiene 192 núcleos CUDA de precisión simple los cuales comparten 64 Kb de memoria en chip que se pueden configurar como 48Kb de memoria compartida y 16Kb de cache L1 para aumentar la ocupación en las aplicaciones que utilizan una gran cantidad de memoria compartida, pero también es posible configurar 16/48 KB y 32/32 KB. La primera configuración puede disminuir la latencia de acceso a la memoria a nivel global y la última obviamente equilibra los dos beneficios.

.

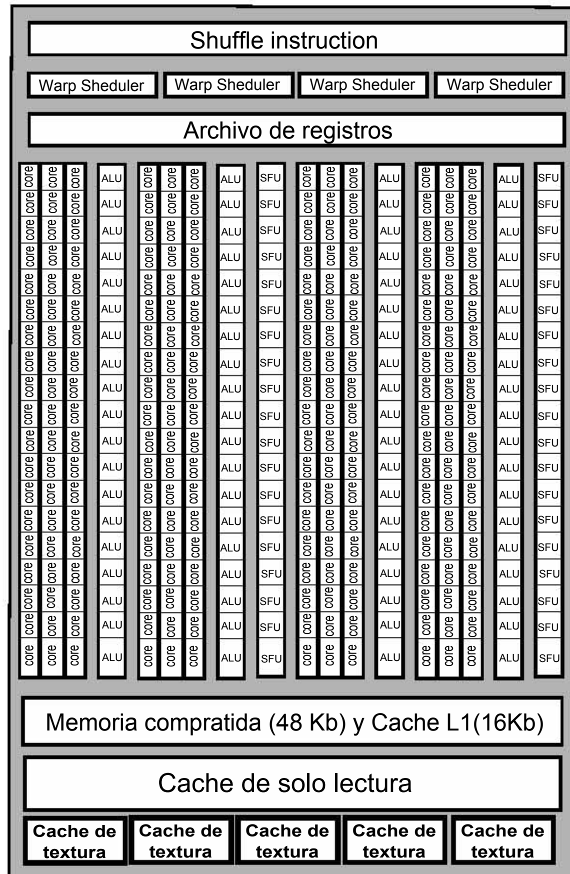


Figura 2.5: Estructura física de un multiprocesador

Sin embargo fueron tres aspectos que hicieron la diferencia con las versiones anteriores de NVIDIA, para mejorar aún más el rendimiento. La arquitectura Kepler implementó una nueva unidad de instrucciones llamada *shuffle instruction* que permite a los hilos contenidos en un *warp* compartir datos. Anteriormente, el intercambio de datos requería la carga y almacenamiento independiente para pasar los datos a través de la memoria compartida.

Otro aspecto relevante es la *programación dinámica*. Anteriormente las Fermi lanzaban todo el trabajo desde la CPU, procesaban los datos en la GPU y el resultado lo enviaba al host para ser analizados y devolverlos al dispositivo o para usarlos como solución final; ahora un *kernel* puede generar otro *kernel* por si solo sin las necesidad de crearlos desde la CPU (paralelismo anidado). Esto permite la ejecución de algoritmos jerárquicos. Por ejemplo, en CDF se suele crear una malla en las regiones donde se manifiesta el fenómeno con el fin de distribuir el trabajo uniformemente en los procesadores, pero la naturaleza de los fenómenos no siempre permite una carga equilibrada; la programación dinámica permite dividir aún

.

más la malla creando *kernels* anidados, hasta 24 niveles de profundidad, en las zonas con más carga computacional.

La tercera característica que vale la pena resaltar es a lo que NVIDIA llama *Hyper-Q* que permite la conexión simultánea, entre el host y el distribuidor de trabajo de CUDA (CWD), de 32 streams CUDA como procesos de MPI o subprocesos dentro de un proceso evitando los cuellos de botella que se generaban a causa de la única conexión que existía en las tarjetas Fermi. Ahora cada proceso tiene su propia conexión y ya no existe la necesidad de especificar el orden de lanzamiento para evitar falsas dependencias entre procesos MPI (NVIDIA, 2014).

2.2.2 *Xeon Phi*

Esta tarjeta se considera como un coprocesador porque está diseñado para parecer y operar como un procesador multinúcleo común, sólo que con más núcleos. Un IXPC (Intel Xeon Phi Coprocesor) procesa vectorialmente de manera natural instrucciones de 512 bits y realiza ocho o dieciséis operaciones en simple y doble precisión, respectivamente, en sus 61 núcleos. Cada núcleo tiene una cache L1 de datos e instrucciones de 32Kb cada una y una cache L2 de 512Kb y no comparten la cache entre núcleos.

Cuando se tiene un dominio de simulación es posible dividirlo en 60 tareas con MPI y asignarlo a cada núcleo para después ejecutarlo con hasta cuatro hilos OpenMP que puede almacenar cada núcleo. La biblioteca Intel MPI explota el paralelismo en modelos de memoria distribuida, por lo que es posible manejar los coprocesadores como nodos individuales, puesto que cuenta con puerto ethernet, acceso SSH y además es posible asignar una dirección IP.

El IXPC maneja dos modelos de programación; el modelo *offload* considera el intercambio de mensajes entre el acelerador y el host. Este modelo está orientado al paralelismo OpenMP entre núcleos por lo que se debe evitar la creación de regiones paralelas se debe reducir al mínimo en el ciclo del tiempo, además, implementa el *multitheading* para manejar hilos OpenMP, sólo que no lo hace explícitamente, el programador debe crear suficientes hilos para poder soportar la latencia. De igual manera se pueden realizar operaciones SIMD por hilo, puesto que cuenta con directivas que controlan la vectorización.

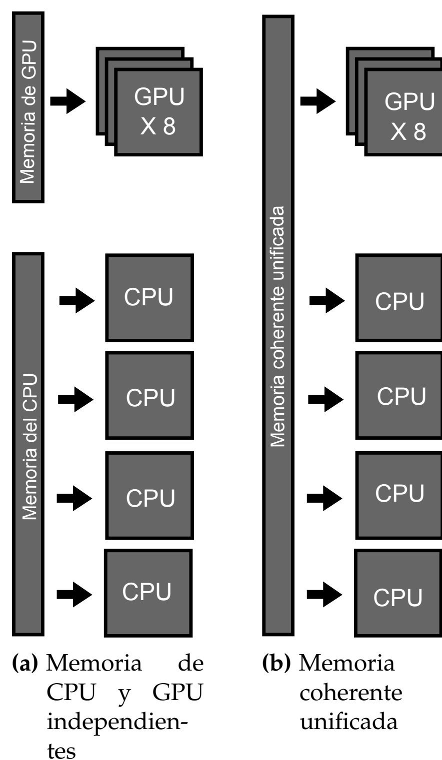
El modelo MPI mantiene la comunicación entre nodos cuando hay más de una tarjeta IXPC. Se puede dar el caso de tener procesos distribuidos entre el host y el coprocesador de la misma manera como trabaja cualquier *cluster* o mantenerlos de forma nativa en el coprocesador.

.

2.2.3 *AMD*

AMD implementó una tecnología llamada *núcleos de cómputo* que permite que los núcleos de CPU y GPU compartan la memoria y la carga de trabajo, haciendo que la comunicación entre los dos sea más rápida y eficiente. Esto es posible porque los cuatro núcleos CPU y los ocho núcleos GPU que componen esta tarjeta se conectan físicamente a una memoria que les permite el acceso a los procesos en un mismo nivel. A esta tecnología le denominaron *acceso a la memoria heterogénea uniforme* o HUMA.

El inconveniente de tener un mismo nivel de asignación y ejecución de tareas, hablando de ambos tipos de núcleos, se presenta al momento de gestionar las tareas cuando existe igualdad de condiciones y carga de trabajo. Para esto, AMD agrega a su tecnología la *cola heterogénea* o HQ por sus siglas en inglés, que evalúa las tareas para después asignarlas al mejor núcleo disponible.



(a) Memoria de CPU y GPU independientes

(b) Memoria coherente unificada

Figura 2.6: Arquitectura APU (*Access Process Unit*)

2.3 El modelo de programación de OpenMP

OpenMP es una especificación que define un conjunto de directivas o del inglés *pragma*. OpenMP no es un lenguaje de programación, es un conjunto de directivas y funciones que en conjunto con las variables de ambiente permiten al programador expresar el paralelismo en máquinas de memoria compartida en lenguajes como FORTRAN, C y C++.

Cuando se insertan las directivas en el código fuente hacen que el compilador genere código que puede ser ejecutado en paralelo utilizando múltiples hilos de ejecución. En general no se necesita utilizar demasiadas directivas o modificar la estructura del código, por tal motivo OpenMP se ha convertido en una especificación bastante aceptada por los principales proveedores de equipos de alto rendimiento como Oracle, Intel, IBM y SGI, ya que de manera uniforme y transportable se puede producir programas que se ejecutan en paralelo.

2.3.1 *Modelo de ejecución de OpenMP*

Después de una directiva `BARRIER`, `PARALLEL` y `FOR` existe implícitamente una directiva `FLUSH`

La API OpenMP utiliza el modelo de ejecución Fork-Join que consiste en dividir una tarea en subtareas para ser ejecutadas por un conjunto de hilos, una vez finalizadas llegan a un punto de encuentro donde se puede hacer operaciones de reducción llamada barrera. Las tareas se definen explícitamente o implícitamente por las directivas OpenMP. Una tarea es implícita cuando los hilos se comportan de algún modo sin necesidad de especificar alguna directiva. Un caso claro es cuando se genera una barrera con la directiva `BARRIER` antes de sincronizar los valores de los hilos con la memoria para asegurar la consistencia de datos en la memoria antes de hacer alguna configuración previa a variables visibles, de manera que para el sistema es trasparente generar la sinconización de memoria sin la necesidad de colocar explicitamente la directiva `FLUSH`.

Un programa OpenMP comienza ejecutando un hilo inicial o maestro secuencialmente hasta encontrar un constructor paralelo donde se crea un conjunto de hilos que ejecutan el mismo código pero sobre diferentes datos por lo que se ejecutan de modo SIMD. Al final de la región paralela existe una barrera implícita donde se sincronizan y únicamente el hilo maestro continua la ejecución del código posterior a la región paralela (ver Figura 2.7).

Se pueden abrir tantas regiones paralelas como sean necesarias para optimizar el código, incluso la API de OpenMP permite anidar regiones paralelas controladas por las variables de entorno OMP_NESTED = TRUE o FALSE que activa o desactiva la función de crear nuevas tareas y asignarlas a nuevos hilos esclavos,

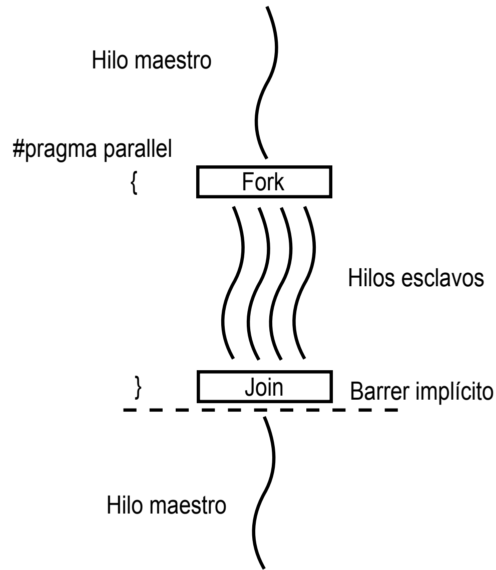


Figura 2.7: Modelo de ejecución Fork-Join de OpenMP

generados desde grupos de hilos de un nivel arriba (ver Figura 2.8). El correcto funcionamiento de esta configuración depende en gran parte del compilador porque la API de OpenMP no lo obliga a soportar esa configuración.

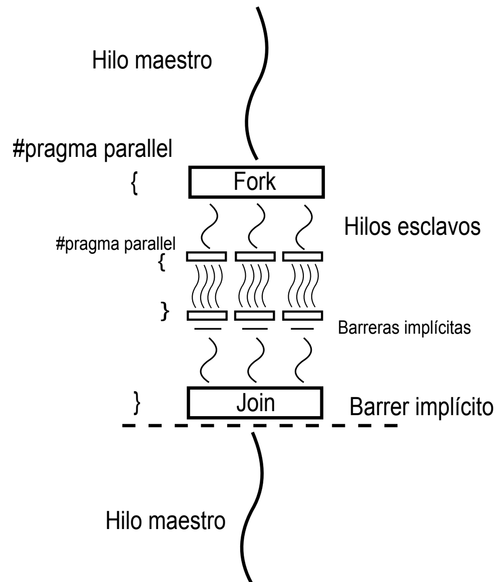


Figura 2.8: Modelo de ejecución Fork-Join en regiones paralelas anidadas

2.3.2 *Modelo de memoria OpenMP*

El modelo de memoria de OpenMP está enfocado a la forma en la cual los hilos tendrán acceso a las variables, el grado de optimización y la disminución de tiempo de latencia dependen del entendimiento y de la buena planeación del programador. El sistema permite que todos los hilos tengan un lugar para almacenar y recuperar las variables, generalmente es en la memoria principal pero es posible que cada hilo tenga su propia instancia temporal de rápido acceso, cuando hablamos de rápido acceso a espacios de memoria generalmente estamos hablando de registros, memoria cache u otro almacenamiento intermedio.

Cada hilo tiene acceso privado a un tipo de memoria conocida como *thread-private* como se muestra en la Figura 2.9, que no es accesible desde otro hilo. Esta memoria almacena las instancias de la variable cuando se configuran los atributos de intercambio de datos como privados (`PRIVATE`). Esta configuración crea una nueva versión de la variable, de mismo tipo y tamaño, para cada tarea especificada en el bloque de código estructurado asociado a esa directiva.

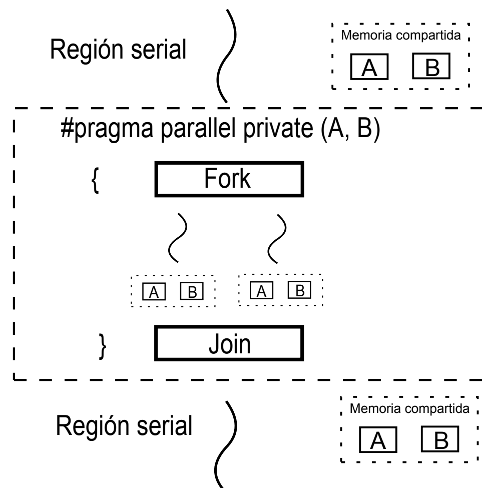


Figura 2.9: Comportamiento de la memoria en una variable privada

Otra configuración (ver Figura 2.10) de intercambio de datos es aquella que al momento de acceder a ella en el bloque estructurado, se hace referencia a la variable original. Esta configuración es conocida como de acceso compartido especificado por la cláusula `SHARED`. Una variable especificada como compartida puede ser accedida por múltiples hilos que pueden ver el contenido de las tareas, cambiar su contenido y ver los cambios realizados de manera instantánea, a esto se le conoce como *cache coherency*. Es conveniente programar la sincronización de lectura y escritura para evitar condiciones de carrera; estas ocurren cuando un hilo lee un espacio de memoria y otro escribe en ese mismo espacio produciendo resultados inesperados.

.

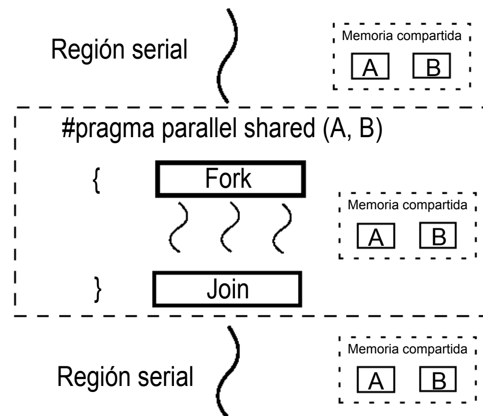


Figura 2.10: Comportamiento de la memoria en variables compartidas

2.3.3 *Comandos de interés para el desarrollo de programas en OpenMP*

La directiva `PARALLEL` marca el inicio de la parte fork del modelo de ejecución, en este punto se crean una serie de hilos paralelos que comienzan inmediatamente la ejecución redundante de las instrucciones que contiene la directiva. Inmediatamente después se declaran las variables que usarán los hilos (el compilador tiene la capacidad de decidir la forma en la que serán accedidas). Sin embargo, el programador tiene el control total si se declara la clausula `DEFAULT (NONE)` y especificando explícitamente el acceso con las clausulas `SHARED, PRIVATE ,FIRSTPRIVATE,` etc.

Para completar la sintaxis de la directiva `PARALLEL` se debe especificar el número de hilos y la forma en la que se ejecutarán paralelamente. El número de hilos dentro de una región paralela puede ser especificado de diferentes maneras, una a través de la variable de ambiente `OMP_NUM_THEADS`, otra por medio de la rutina `OMP_SET_ NUM_THREADS()`.

Cuando un hilo alcanza un bucle de trabajo compartido, ese hilo ejecutará un subconjunto de instrucciones contenidas en el ciclo en función a la planificación de trabajo especificado explícitamente por el programador con la cláusula `SHEDULE`. Esta última cláusula no es necesaria porque el compilador puede determinarla de forma implícita, al final del ciclo, los hilos se sincronizan en una barrera al menos que el usuario haya especificado la clausula `NOWAIT`.

Las variables de entorno `OMP_GET_THREAD_NUM() OMP_GET_NUM__THREAD` y nos ayudarán a saber el número de hilo y la cantidad de hilos que se están ejecutando respectivamente. Estas herramientas son de gran utilidad al momento de depurar el código o en su defecto para asignar tareas a un hilo en especifico. La directiva

Código 1: Clausula REDUCTION

```
!$OMP DO COLLAPSE(2) REDUCTION(+:ETOT)
        DO i = 2, (ni-1)
                DO j = 2, (nj-1)
                ETOT = ETOT + ET(i,j)*VOL(i,j);
                END DO
        END DO
!$OMP END DO
```

`!$OMP SINGLE` también controla la ejecución ordenada de hilos. Esta directiva es útil cuando se requiere que las instrucciones sean ejecutadas por un solo hilo que en lo general es el que llegue primero a esta zona, no importa cual sea. Por otro lado la directiva `!$OMP TASK` ejecuta las instrucciones por un hilo sin importar el orden de la ejecución, su función se detalla más a fondo.

La ejecución del código intensivo `!$OMP DO` puede estar modificado por las clausulas `COLLAPSE` Y `REDUCTION`. La distribución de iteraciones entre los hilos en ciclos convencionales consiste en asignar una iteración del ciclo exterior e iterar el ciclo interior de forma serial, pero al colapsar los ciclos se aumenta la granularidad y por ende el número de iteraciones; con esto es posible optimizar el código cuando la vectorización no es aplicable. Por otro lado para el control de instrucciones de reducción como el fragmento de Código 1, la opción para lograr la suma es necesaria la cláusula `REDUCTION`, sólo es necesario especificar la variable y la operación reductiva.

Por último, OpenMP proporciona una función en tiempo de ejecución que contabiliza en segundos el tiempo de cómputo, la sintaxis es la siguiente `!$OMP_GET_WTIME`.

OpenMP por sus facilidades para generar código multi-hilado se ha convertido en un estándar para la industria y podemos listar sus características relevantes:

- Los códigos con directivas OpenMP solamente se ejecutan en paralelo en máquinas de memoria compartida, aunque existen algunos compiladores que toman las directivas OpenMP y las convierten al modelo de paso de mensajes para su ejecución distribuida.
- Es altamente portátil entre plataformas, por lo que no es necesario cambiar el código fuente para recompilarse y ejecutarse en diferentes arquitecturas.
- Produce tareas de granularidad media y fina. Las tareas son llevadas a cabo por los hilos
- Es un modelo de paralelismo implícito.

.

- Provee una mayor abstracción computacional que los modelos basados en paso de mensajes como MPI.

2.4 El modelo de programación de OpenACC

Debido al rendimiento que se tiene por *watt*, las arquitecturas de HPC se han migrado a un modelo de hardware híbrido combinando arquitecturas multi-núcleo con multi-GPU. No obstante la programación en aceleradores gráficos de procesamiento general a bajo nivel puede complicar el diseño y alargar los tiempos de programación; más aún si no es experto en la arquitectura de la tarjeta, se puede desarrollar un código que al ejecutarse genere un rendimiento menor al esperado conduciendo a un proceso improductivo de desarrollo propenso a errores y demasiado apegado a la arquitectura de la tarjeta, lo cual no es aceptado para proyectos de desarrollo con un ciclo de vida largo.

Los desarrolladores de la tecnología OpenMP publicaron en noviembre del 2011 el estandar OpenACC basado en directivas con el fin de unificar la sintaxis para aceleradores y hacerla disponible entre varias arquitecturas. Dada la problemática de desarrollo de *kernels* propensos a errores se pensó en dar al compilador la encomienda de generar los *kernels* de bajo nivel que tendría que generar el programador manualmente, simplificando la programación en los aceleradores, mejorando la productividad en el desarrollo y simplificando el mantenimiento del código.

Puede inferirse que OpenACC nace de la necesidad de tener un paradigma de programación general para las arquitecturas basadas en aceleradores. No obstante, los modelos de programación más utilizados como CUDA y OpenCL requieren de la creación de funciones *kernels* a bajo nivel por parte del programador; CUDA está acoplado a los GPUs NVIDIA, mientras que OpenCL pretende ser un estándar portable a través de distintas arquitecturas de hardware; sin embargo el uso de APIs a bajo nivel puede resultar en una implementación tediosa de código y propensa a errores, y con OpenACC mitigamos esta tarea haciendo al compilador responsable.

2.4.1 *Modelo de ejecución de OpenACC*

El modelo de ejecución que implementa la API de OpenACC se conoce como host-device. El código especificado fuera de las regiones paralelas es ejecutado por el *host* y las regiones de cálculo intensivo, en su mayoría sentencias de repetición, son cargadas por el *host* al dispositivo.

El *host* es el responsable de asignar la memoria en el dispositivo, sin embargo no es capaz de acceder a ella directamente. Una vez asignada la memoria, el código que conforma los *kernel* es trasferido al dispositivo, de igual manera el *host* es el que alimenta de argumentos a la región de cálculo. Una vez iniciada la ejecución de los *kernels* el dispositivo espera la ejecución total de los mismos para trasferir el código procesado de nuevo al *host* y finalmente libera la memoria.

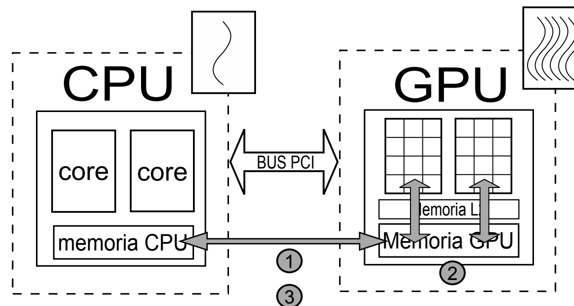


Figura 2.11: 1. Se copian los datos de la memoria del CPU a la memoria del GPU, 2. Lee el programa GPU y carga los datos a Core CUDA para procesarlos, por ultimo 3. Se copian los resultados desde la memoria del GPU a la memoria del CPU.

OpenACC está diseñado para soportar tres niveles de paralelismo, pero depende del entendimiento del programador aplicar y explotar sus beneficios. En la mayoría de las investigaciones consultadas para desarrollar el presente trabajo, paralelizaban en su mayoría ciclos `FOR` o `DO`.

En un primer nivel la API divide la carga de trabajo entre las unidades de ejecución. Estos bloques se conocen como *gangs*, obteniendo así, un paralelismo de grano grueso. El paralelismo de grano fino es representado por los *workers*, al igual que los *gangs*, los *workers* son grupos de subprocesos o hilos concentrados en múltiplos de *warps*, pero son distribuidos entre los núcleos que conforman una unidad de ejecución. Por último, la ejecución SIMD o `vector` por parte de cada hilo depende del acelerador, ya que no todos los dispositivos lo soportan. Dentro de cada *gang*, las instrucciones son ejecutadas de manera redundante por cada hilo que se ejecuta de manera vectorial (ver Figura 2.12).

Cabe señalar que no es necesario especificar algún tipo de barrera para sincronizar la ejecución de los *gang*, dado que el modelo de ejecución espera hasta que el bloque anterior se haya completado para ejecutar un nuevo *gang*.

En un inicio el programa es ejecutado por un hilo maestro, al igual que la ejecución de OpenMP. Cada hilo en ejecución almacena un identificador de hilo y el número de bloque que lo contiene, algo muy similar a la identificación que maneja CUDA. OpenMP permite generar nuevos hilos desde el host a partir del hilo maestro que, en conjunto con la identificación de hilos, nos permite distribuir

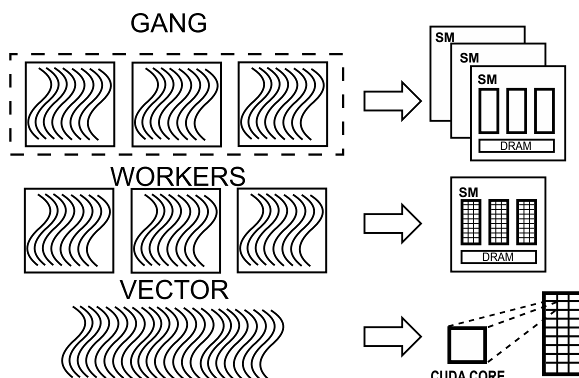


Figura 2.12: OpenACC considera tres niveles de paralelismo: gangs, workes y vector

el trabajo no sólo entre unidades de ejecución, sino también entre dispositivos si se cuenta con más de una GPU. Esta teoría fue aplicada en la investigación de Rengan Xu, S. Chandrasekaran y B. Chapman en la cual se logró aumentar la velocidad hasta dos veces comparada con la ejecución con una sola GPU (Xu et al., 2015).

En algunos dispositivos, como en las tarjetas NVIDIA, el compilador también es capaz de crear y lanzar *kernels* desde otro *kernel* lo que permite el paralelismo anidado.

2.4.2 *Modelo de memoria de OpenACC*

OpenACC supone que la memoria del acelerador y del host son totalmente independientes, por lo que cada hilo host no puede leer o escribir directamente en la memoria del dispositivo. Bajo estas condiciones no se puede manejar de forma transparente la existencia de varias instancias de una variable de *host* en el dispositiv;, además, es importante mencionar que las referencias a espacios de memoria en el *host* o en el dispositivo sólo son válidas en el lugar donde se referencia.

Los movimientos de datos entre la memoria del dispositivo y la memoria del *host* son llevadas acabo por el hilo principal, al igual que en CUDA los datos de entrada deben ser transferidos antes de ejecutar el *kernel* y los datos resultantes son devueltos desde el dispositivo al *host*. El compilador de OpenACC tiene la ventaja de generar automáticamente el código necesario para asignar, copiar y liberar memoria, lo que lo vuelve más sencillo en comparación al manejo explicito de los datos en códigos CUDA a base de *kernels*.

Las directivas son la forma en la que el programador indica cómo será ejecutado el código. Las más importantes son `KERNEL` Y `PARALLEL`, puesto que el compilador

.

interpreta que tiene que crear un *kernel* CUDA para ese código. Si no se especifica alguna cláusula adicional que indique la forma en la que será ejecutado el *kernel*, entonces se elige en tiempo de ejecución una configuración que eleve el rendimiento. Estas configuraciones por defecto suelen ser benéficas en la mayoría de los casos.

Cuando el compilador detecta la directiva `kernel` y no genera el código necesario para ejecutarse en el dispositivo, es posible que al haber hecho su análisis el compilador detecta dependencias de flujo. Sin embargo podemos indicar al compilador con las cláusula `independent` que pase por alto su propio análisis de dependencia y confíe en el programador, puesto que detecta que no existe ninguna dependencia.

Entre las cláusulas que nos ayudan a especificar la ejecución y la gestión de datos de manera explícita se encuentran `copyin` y `copyout`, las cuales especifican el tipo y dirección en el movimiento de datos. También es posible crear variables exclusivas del dispositivo con `CREATE` o indicar al compilador que los datos ya se encuentran en el dispositivo con la clausula `PRESENT`. La coherencia de datos entre el *host* y el dispositivo puede controlarse explícitamente con la cláusula `UPDATE` para sincronizar ambas memorias y no perder la consistencia de datos.

Por último, la directiva `DATA` juega un papel importante en el aumento de rendimiento, la principal ventaja de su uso es la reutilización de datos, muy útil cuando se paralelizan ciclos anidados porque ocurren múltiples trasferencias entre el *host* y el dispositivo causando una perdida de rendimiento.

Las características relevantes de OpenACC son:

1. Las directivas proporcionan una forma relativamente fácil para acelerar aplicaciones intensivas en cómputo.
2. Las directivas de OpenACC son abiertas y estandarizadas, generando códigos paralelos de una forma más sencilla y portable entre plataformas multinúcleo.
3. Permite acceder de manera transparente al cómputo masivo en GPUs

.

Capítulo 3

El esquema numérico del caso de estudio

Un fluido es considerado como un conjunto de moléculas dispersas con un espacio vacío entre ellas (Anderson and Wendt, 1995). Para simular su comportamiento se debe aplicar los tres principios físicos fundamentales: la fuerza representada por la segunda ley de Newton, la conservación de la masa y la conservación de la energía. Cabe señalar que aunque describen cualquier fluido en movimiento la forma en la que se plantea puede llevar a inestabilidad, resultados incorrectos u oscilaciones en los resultados numéricos.

El modelo matemático puede ser planteado desde dos perspectivas: En principio se considera un segmento finito F del dominio delimitada por una superficie cerrada S por el cual el flujo se mueve, como si fuera la ventana de una casa en la que se observan los autos pasar sobre la avenida; el segundo caso considera el mismo segmento F pero moviéndose con el flujo conservando las partículas que contiene. Volvamos al ejemplo anterior pero ahora tomaremos los autos visibles a través de ella y observamos su comportamiento a lo largo de la avenida.

Estos modelos pueden ser representados en forma de integrales y trasformarlas a ecuaciones diferenciales parciales; las ecuaciones que representan el modelo donde el volumen se mueve con el fluido se conocen como de no conservación y para el caso donde el volumen es fijo se conoce como ecuaciones de conservación. Otro elemento importante para el modelo es la diferencial del volumen dv, representado por un elemento infinitesimal del fluido lo suficientemente grande para contener una gran cantidad de moléculas; este elemento se define de acuerdo con los dos modelos anteriores. Para el modelo de conservación el elemento se mantiene constante mientras que para el modelo de no conservación el elemento se mueve a lo largo del vector velocidad V.

El elemento infinitesimal en el modelo de no conservación nos ayudará a comprender el concepto de *derivada material*, empleado en aerodinámica, pero desde el punto de vista físico. Para hacer más gráfica la explicación en la Figura 3.13 muestra el elemento infinitesimal moviéndose a través de un plano cartesiano donde el vector velocidad está dado por:

$$v = u_i + v_j + w_k \tag{1}$$

.

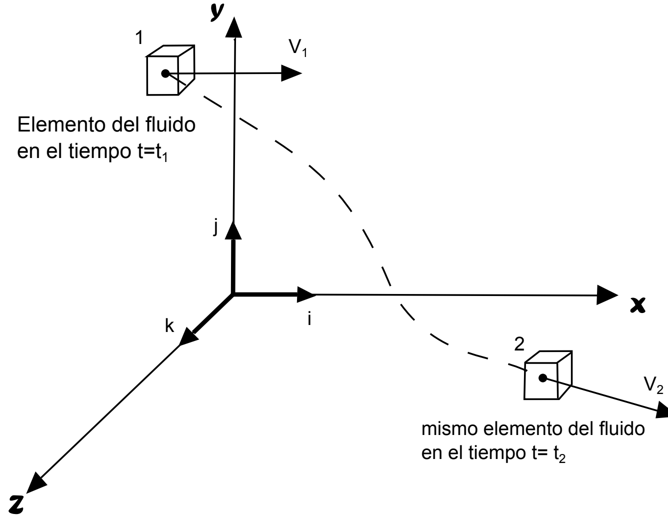


Figura 3.13: Movimiento infinitesimal

Al considerar un fluido no constante, los componentes u, v y w estarán en función del espacio y el tiempo expresados como sigue $u = u(x, y, z, t)$, $v = v(x, y, z, t)$ y $w = w(x, y, z, t)$ y el campo de densidad escalar está dada por $\rho = \rho(x, y, z, t)$.

Si observamos la Figura 3.13 encontramos que el elemento infinitesimal está situado en el punto 1 en el tiempo t_1, en ese momento la densidad es $\rho_1 = \rho(x, y, z, t)$ pero si la partícula se mueve su densidad cambia, de modo que en el punto 2 en el tiempo t_2 la densidad es $\rho_2 = \rho(x, y, z, t)$. Dado que $\rho = \rho(x, y, z, t)$ podemos aproximar la función a partir de las dos densidades con una serie de Taylor de la siguiente manera:

$$\rho_2 = \rho_1 + \left(\frac{\partial \rho}{\partial x}\right)_1 (x_2 - x_1) + \left(\frac{\partial \rho}{\partial y}\right)_1 (y_2 - y_1) + \left(\frac{\partial \rho}{\partial z}\right)_1 (z_2 - z_1) + \\ + \left(\frac{\partial \rho}{\partial t}\right)_1 (t_2 - t_1) + (\text{términos de mayor orden}) \quad (2)$$

Dividiendo entre $(t_2 - t_1)$ e ignorando los términos de orden mayor obtenemos

$$\frac{\rho_2 - \rho_1}{t_2 - t_1} = \left(\frac{\partial \rho}{\partial x}\right)_1 \frac{x_2 - x_1}{t_2 - t_1} + \left(\frac{\partial \rho}{\partial y}\right)_1 \frac{y_2 - y_1}{t_2 - t_1} + \\ + \left(\frac{\partial \rho}{\partial z}\right)_1 \frac{z_2 - z_1}{t_2 - t_1} + \left(\frac{\partial \rho}{\partial t}\right)_1 \quad (3)$$

Si observamos el lado izquierdo de la ecuación, la expresión representa físicamente la razón de cambio promedio respecto al tiempo de la densidad cuando se

.

mueve del punto 1 al punto 2.

Por consiguiente, si aplicamos el limite de t_2 cuando tiende a t_1 en ambos lados de la ecuación podríamos simplificar esa expresión, además la razón de cambio de la densidad con respecto al tiempo representa el movimiento de un elemento infinitesimal a través del espacio.

$$\lim_{t_2 \to t_1} \frac{\rho_2 - \rho_1}{t_2 - t_1} \equiv \frac{D\rho}{Dt}$$

Los términos del lado derecho de la ecuación (3) quedan de la siguiente manera:

$$\lim_{t_2 \to t_1} \frac{x_2 - x_1}{t_2 - t_1} = u$$

$$\lim_{t_2 \to t_1} \frac{y_2 - y_1}{t_2 - t_1} = v$$

$$\lim_{t_2 \to t_1} \frac{z_2 - z_1}{t_2 - t_1} = w$$

El resultado son los componentes de la velocidad por lo tanto es posible reescribir la ecuación (3) de la siguiente manera:

$$\frac{D\rho}{Dt} = u\frac{\partial \rho}{\partial x} + v\frac{\partial \rho}{\partial y} + w\frac{\partial \rho}{\partial z} + \frac{\partial \rho}{\partial t} \tag{4}$$

A partir de la ecuación anterior podemos obtener una expresión para la derivada material en coordenadas cartesianas

$$\frac{D}{Dt} \equiv \frac{\partial}{\partial t} + u\frac{\partial}{\partial x} + v\frac{\partial}{\partial y} + w\frac{\partial}{\partial z} \tag{5}$$

Finalmente, en coordenadas cartesianas el vector gradiente se define como

$$\nabla \equiv i\frac{\partial}{\partial x} + j\frac{\partial}{\partial y} + k\frac{\partial}{\partial z} \tag{6}$$

Así que la ecuación (5) la podemos definir como

$$\frac{D}{Dt} \equiv \frac{\partial}{\partial t} + (v \cdot \nabla), \tag{7}$$

donde $\frac{D}{Dt}$ no es más que una derivada total, la cual es físicamente la tasa de cambio con respecto al tiempo en un punto fijo; por otro lado $v \cdot \nabla$ es la derivada direccional y físicamente representa la tasa de cambio con respecto al tiempo ocasionada por el movimiento del elemento infinitesimal en el espacio donde las propiedades del fluido son espacialmente diferentes.

3.1 Variables y parámetros

En el campo de la hidráulica el análisis de los fluidos se centra en las condiciones físicas como la densidad, la presión o la temperatura que se manifiesta a

.

nivel macroscópico. Sin embargo, no se estudia a las moléculas como ente individual por la complejidad que esto conlleva, por ello, se recurre al supuesto de la distribución continua de materia a la que se denomina como medio continuo.

3.1.1 *Propiedades de los fluidos*

Densidad ρ. Considerando sus unidades en el sistema internacional ($Kg/m3$) podemos definir a la densidad como la cantidad de masa que existe por cada unidad de volumen. Si se supone un fluido homogéneo entonces se podría calcular como se muestra en la siguiente ecuación, puesto que la densidad no varía de un punto a otro.

$$\rho = \frac{V}{m} = \frac{Volumen}{masa}$$

Para el caso de la presente aplicación se tomará la densidad del agua como 1.000 kg/m^3 a 4°C según el sistema internacional (SI).

El peso espec/'ifico(γ) de un fluido se puede calcular de dos formas: si se conoce la densidad ρ y tomando en cuenta la constante de gravedad g, entonces se puede aplicar la ecuación

$$\gamma = \rho g,$$

pero si el fluido se encuentra a condiciones específicas de presión *p* y temperatura *T* se podría calcular con la ecuación

$$\gamma = \frac{p}{RT},$$

donde R es la constante del gas que se trata. Sus unidades según el SI son N/m^3, por lo que el peso específico se podría definir como el peso por unidad de volumen. Para el caso del agua el peso específico es 1.000 kg/cm^3.

Viscosidad. La característica principal de los fluidos es la fluidez, se denomina así al esfuerzo cortante que se aplica al fluido para que se produzca un cambio continuo en él, más concretamente es la fuerza que provoca que el fluido se mueva. La rapidez con que este se mueva va a depender de la viscosidad. L. Mott (Mott, 1996) define la viscosidad como *la propiedad de un fluido que ofrece resistencia al movimiento relativo de sus moléculas*.

Un fluido no es capaz de soportar un esfuerzo cortante sin moverse a diferencia de un sólido que sí puede, Figura 3.14. Si consideramos dos láminas, una fija y otra en movimiento a una velocidad ν entonces el fluido entre ellas tendrá la misma velocidad que la frontera. El cambio de velocidad estará representado por un elemento de fluido ($\frac{\partial u}{\partial y}$) que se conoce como gradiente de velocidad y el esfuerzo cortante (τ) representa la fuerza requerida para deslizar una lámina sobre otra del

.

mismo material, sus unidades son Newtons sobre metro cuadrado (N/m^2) y se calcula con la siguiente expresión:

$$\tau = \mu \frac{\partial u}{\partial y},$$

donde la letra griega μ representa la constante de proporcionalidad conocida como viscosidad dinámica del fluido.

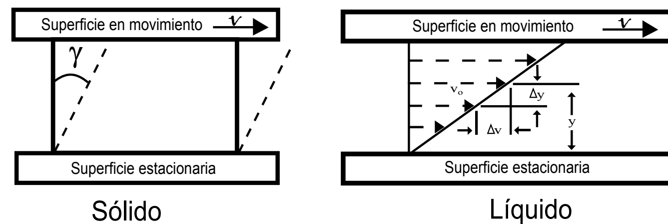


Figura 3.14: Ilustra la viscosidad lineal ν a través del canal y el esfuerzo de corte τ

La presión se define como la fuerza ejercida sobre un área. Sus unidades son los Pascales gracias a las aportaciones del científico Blaise Pascal, sus leyes se resumen en dos principios. El primero nos dice que, dado un pequeño volumen definido de fluido, la presión actual desde él en todas direcciones y la segunda nos sitúa entre fronteras solidas donde la presión del fluido es perpendicular a la superficie Figura 3.15.

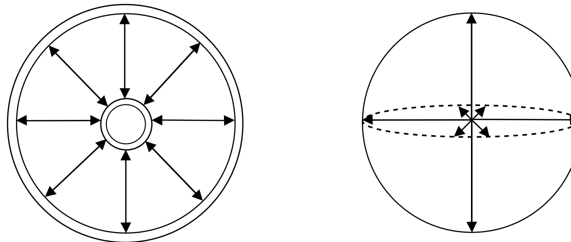


Figura 3.15: Fuerza de presión

3.1.2 *Tipo de fluidos*

Los fluidos se clasifican por su comportamiento en condiciones variadas de temperatura, presión y velocidad, entre otras, y de eso depende su estudio.

Flujo isoentrópico. Cuando un fluido está en constante cambio, la fricción y la transferencia de calor se vuelven despreciables por lo que se supone que entre el medio y el fluido no existe transferencia de calor y es posible estudiar a estos fluidos como flujos isoentrópicos (Anderson and Wendt, 1995).
Flujos subsónicos y supersónicos. Para explicar este tipo de fluido se hablará de un

índice llamado número de Mach que requiere de la velocidad del sonido para su cálculo.

$$M = \frac{V}{C} \tag{8}$$

donde V es la velocidad local y C es velocidad del sonido en el medio.

La velocidad del sonido les da su nombre; los que viajan a velocidades mayores son llamados supersónicos y los que viajan a una velocidad menor se llaman subsónicos. Por lo anterior se infiere que los fluidos con número de Mach mayor a 1 son supersónicos y subsónicos a los fluidos con M<1. Puede darse el caso de que un cuerpo tenga las tres posibles magnitudes de Mach ($M < 1$, $M > 1$ y $M = 1$) en diferentes partes del cuerpo al mismo tiempo; este tipo de fluido cambia su nombre a transónico (tal situación se debe simplemente a que las velocidades varían).

Flujo compresible e incompresible. La diferencia entre un gas y un líquido, sabiendo que los dos son fluidos, es la capacidad de compresión. El término compresible o incompresible depende de la alteración de la densidad por efectos de la temperatura y la presión. Con frecuencia los líquidos y sólidos se consideran incompresibles por la gran cantidad de presión que se necesita para lograr un cambio significativo en su densidad. Los gases, por el contrario, relacionan su densidad con la temperatura y la presión por medio de la ley de los gases mostrada en la ecuación (9) y por eso se consideran compresibles, sin embargo, bajo condiciones especiales de presión (variaciones pequeñas) se puede considerar incompresible.

$$P = \rho RT. \tag{9}$$

De esta última aseveración podemos agregar que a velocidades supersónicas acompañadas de un número de Mach pequeño el gas sometido a esas condiciones es considerado incompresible.

3.2 Ecuaciones que gobiernan el flujo

El fluido que simula el modelo matemático corresponde a un flujo viscoso que viaja a velocidad supersónica y además no presenta difusión de masa, esto quiere decir que hay concentración de gradientes de diferentes especies químicas en el flujo.

El vector que describe el fluido bidimensional tiene dos componentes de velocidad (u y v) y dos propiedades termodinámicas que además no cambian con el tiempo, por lo que se considera un flujo estable. Las ecuaciones de Navier-Stoker se aplican en una amplia variedad de procesos industriales y simulación de flujos

geofísicos. Para el caso particular del fluido dentro de un eyector se considera la ecuación bidimensional de Navier-Stoker escrita en su forma conservativa.

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} = 0. \tag{10}$$

El vector densidad U y los vectores columna F y G que modelan las variables de flujo están definidos de la siguiente manera.

$$U = \begin{cases} \rho \\ \rho u \\ \rho v \\ \rho e \end{cases}$$

$$F = \begin{cases} \rho u \\ \rho u v + p - \tau_{xx} \\ \rho u v - \tau_{xy} \\ \rho u \left(e + \frac{u^2+v^2}{2}\right) + pu - u\tau_{xx} - v\tau_{xy} + q_x \end{cases}$$

$$G = \begin{cases} \rho v \\ \rho u v - \tau_{xy} \\ \rho v^2 + p - \tau_{yy} \\ \rho v \left(e + \frac{u^2+v^2}{2}\right) + pv - u\tau_{xy} - v\tau_{yy} + q_y \end{cases}$$

donde

ρ	densidad
v ,u	componentes de la velocidad
p	presión termodinámica
e	energía interna especifica
τ_{xx} , τ_{yy} y τ_{xy}	componentes del tensor de esfuerzo de viscosidad newtoneana
q_x y q_y	componentes del flujo de calor difusivo

La energía interna e se puede expresar en términos de u,v,p , ρ y γ que representa al exponente isentrópico, como sigue:

$$e = \frac{u^2 + v^2}{2} + \frac{p}{(\gamma - 1)\rho} \tag{11}$$

.

Cuando se trabaja con un gas perfecto la expresión anterior puede remplazar a la variable e en los vectores, que a su vez fueron separados por sus elementos para facilitar su manejo en el código.

$$\begin{aligned} U_1 &= \rho \\ U_2 &= \rho u \\ U_3 &= \rho v \\ U_4 &= \rho e \end{aligned} .$$

$$\begin{aligned} F_1 &= \rho u \\ F_2 &= \rho u v + p - \tau_{xx} \\ F_3 &= \rho u v - \tau_{xy} \\ F_4 &= \tfrac{\gamma}{(\gamma-1)} p u + \rho u \tfrac{u^2+v^2}{2} - u\tau_{xx} - v\tau_{xy} + q_x \end{aligned} .$$

$$\begin{aligned} G_1 &= \rho v \\ G_2 &= \rho u v - \tau_{xy} \\ G_3 &= \rho v^2 + p - \tau_{yy} \\ G_4 &= \tfrac{\gamma}{(\gamma-1)} p v + \rho v \tfrac{u^2+v^2}{2} - u\tau_{xy} - v\tau_{yy} + q_y \end{aligned} .$$

Como la viscosidad se define como la rapidez con la que se desplazan los fluidos, los términos de viscosidad artificial integrados en los vectores de flujo F y G están expresados en términos de las derivadas de los componentes de la velocidad:

$$\tau_{xy} = \mu \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right), \tag{12}$$

$$\tau_{xx} = \lambda \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) + 2\mu \left(\frac{\partial u}{\partial x} \right), \tag{13}$$

$$\tau_{yy} = \lambda \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) + 2\mu \left(\frac{\partial v}{\partial y} \right). \tag{14}$$

donde λ se define por $\lambda = \frac{2}{3}\mu$ y μ representa la viscosidad dinámica.

Otro término que se considera en los vectores de flujo es la energía calorífica que se produce como resultado de la fricción del flujo y la superficie sobre la cual se desplaza. La ley de calor de Fourier describe este fenómeno con las siguientes expresiones:

$$q_x = -K \frac{\partial T}{\partial x} \tag{15}$$

.

$$q_y = -K\frac{\partial T}{\partial y} \quad (16)$$

Las variables de viscosidad dinámica μ y la conductividad térmica K dependen de la temperatura calculada por la ley de Sutherland definida como:

$$K(T) = \frac{\mu(T)\gamma R}{(\gamma-1)Pr} \quad (17)$$

que integra en su función la viscosidad μ, la temperatura de Sutherland S y el número de PrandtlPr para los cuales se consideraron los siguientes valores iniciales.

$T_o = 273k$ $\quad$ $\gamma = 1{,}4$

$S = 110{,}5k$ $\quad$ $Pr = 0{,}71$

$\mu = 1{,}68x10^-2$

y $\mu(T)$ definida como:

$$\mu(T) = \mu_0\left(\frac{T}{T_0}\right)^{\frac{3}{2}}\frac{T_0+S}{T+S}. \quad (18)$$

3.3 Transformación curvilínea

Para discretizar el sistema en la malla mostrada en la Figura 3.17. El dominio computacional del eyector D ilustrado en la Figura Figura 3.16 es dividido en celdas en un sistema de coordenadas delimitado por $(\xi, \eta) \in$ que toman los valores contenidos en los intervalos cerrados [0,L] x [0,1.0] respectivamente, por lo que la transformada del plano físico (x, y) al campo computacional de las ecuaciones que gobiernan el flujo se define por:

$$\xi = x \quad (19)$$

$$\eta = \frac{y - y_s(x)}{h(x)} \quad (20)$$

La función $y_s(x)$ describe la superficie inferior del eyector y $y_z(x)$ la superficie superior que se describen en las ecuaciones [19] y [20] de modo que la altura $h(x)$ se calcula con la diferencia entre las dos superficies.

$$h(x) = y_z(x) - y_s(x)$$

.

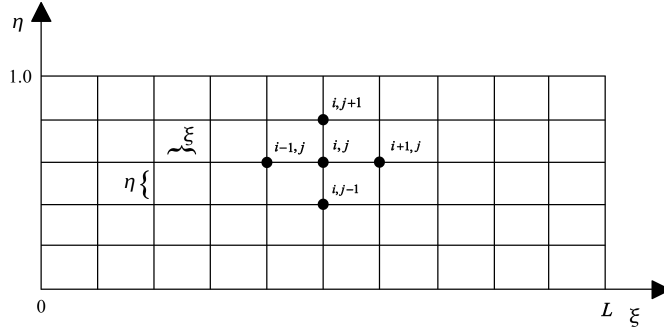


Figura 3.16: Plano computacional.

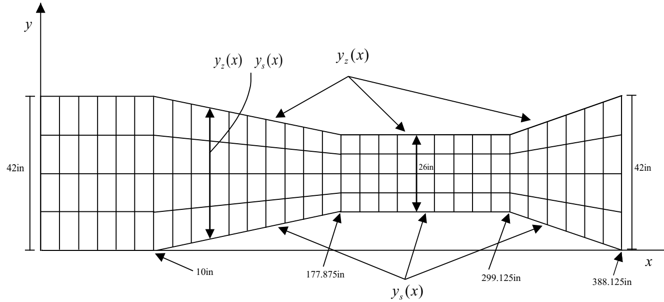


Figura 3.17: Plano computacional.

$$y_s(x) = \begin{cases} 0{,}0 & \text{si} \quad x \leqslant 10{,}0 \\ \frac{8}{167{,}875}(x - 10{,}0) & \text{si} \quad 10{,}0 < x \leqslant 0{,}10 \\ 8{,}0 & \text{si} \quad 177{,}875 < x \leqslant 0{,}10 \\ -\frac{8}{89}(x - 299{,}125) + 8 & \text{si} \quad x > 299{,}125 \end{cases} \tag{21}$$

$$y_z(x) = \begin{cases} 42{,}0 & \text{si} \quad x \leqslant 10{,}0 \\ -\frac{8}{167{,}875}(x - 10{,}0) + 42 & \text{si} \quad 10{,}0 < x \leqslant 0{,}10 \\ 34{,}0 & \text{si} \quad 177{,}875 < x \leqslant 0{,}10 \\ \frac{8}{89}(x - 299{,}125) + 34 & \text{si} \quad x > 299{,}125 \end{cases} \tag{22}$$

Es necesario transformar en términos de ξ y η a la ecuación de movimiento para las dos direcciones x y y por lo que $\frac{\partial}{\partial x}$ y $\frac{\partial}{\partial y}$ quedan de la siguiente manera.

$$\frac{\partial}{\partial x} = \frac{\partial}{\partial \xi}\left(\frac{\partial \xi}{\partial x}\right) + \frac{\partial}{\partial \eta}\left(\frac{\partial \eta}{\partial x}\right) \tag{23}$$

$$\frac{\partial}{\partial y} = \frac{\partial}{\partial \xi}\left(\frac{\partial \xi}{\partial y}\right) + \frac{\partial}{\partial \eta}\left(\frac{\partial \eta}{\partial y}\right) \tag{24}$$

.

Se puede utilizar la regla de la cadena definida en las ecuaciones (23) y (24) para obtener las métricas de la siguiente manera:

$$\frac{\partial \xi}{\partial x} = \frac{\partial \{x\}}{\partial x} = 1 \tag{25}$$

$$\frac{\partial \xi}{\partial y} = \frac{\partial \{x\}}{\partial y} = 0 \tag{26}$$

$$\frac{\partial \eta}{\partial x} = \frac{\partial \left\{ \frac{y - y_s(x)}{h(x)} \right\}}{\partial x} \tag{27}$$

$$\frac{\partial \xi}{\partial y} = \frac{\partial \left\{ \frac{y - y_s(x)}{h(x)} \right\}}{\partial y} \tag{28}$$

despejando y de la ecuación (20) y lo sustituimos en las métricas $\frac{\partial \eta}{\partial x}$ y $\frac{\partial \xi}{\partial y}$, obteniendo:

$$\frac{\partial \eta}{\partial x} = \frac{\left[y_z'(x) - y_s'(x) \right] \eta - y_s'(x)}{h(x)}$$

$$\frac{\partial \eta}{\partial y} = \frac{1}{y_z(x) - y_s(x)}$$

Por lo tanto la ecuación (10) que gobierna el flujo se reformula con las métricas obtenidas de la siguiente manera:

$$\frac{\partial U}{\partial t} = -\left[\frac{\partial F}{\partial \xi} + \frac{\partial F}{\partial \eta} \eta_x \right] - \left[\frac{\partial G_i}{\partial \eta} \eta_y \right] \tag{29}$$

3.4 Condiciones a la frontera

En esta sección hablaremos de las condiciones de frontera aplicadas a la periferia del dominio computacional. Es natural limitar la zona de estudio bajo criterios físicos o lógicos para optimizar recursos computacionales; a esta zona limitada le llamaremos *dominio*. Cuando se simula numéricamente un fenómeno físico que involucre propagación de ondas limitadas por fronteras ficticias, se pueden generan reflexiones, situación que no debería ocurrir, dado que en el dominio real la onda seguiría su curso más allá de la frontera. Una solución sería ampliar el dominio lo suficiente como para que la reflexión no llegue a la zona de interés mientras se cumpla el número de iteraciones. El gran inconveniente de este método es el

.

desperdicio computacional.

La solución más viable para este problema es la técnica PML (Perfectly Match Layer), que consiste en agregar una zona donde se sigan cumpliendo las ecuaciones diferenciales que dominan al flujo, de tal manera que amortigüen la onda.

Para este problema la formulación de la CPML a la salida del flujo fue definida por (Martin and Couder-Castaneda, 2010b), la cual consiste básicamente en reemplazar cada parcial ∂ con $\frac{1}{\kappa}\partial + \Psi$ el avance en tiempo Ψ_x utilizando el mismo esquema de evolución en el tiempo. Si aplicamos esta transformación a las derivadas espaciales a la ecuación principal (ecu. (29)) transformada la ecuación con la formulación de la CPML queda como:

$$\frac{\partial U}{\partial t} = -\left[\left(\frac{1}{k_x}\frac{\partial F}{\partial \xi} + \Psi_{\xi}^{F}\right) + \left(\frac{1}{k_x}\frac{\partial F_i}{\partial \eta} + \Psi_{\eta}^{F_i}\right)(\eta_x)\right] - \left[\left(\frac{1}{k_y}\frac{\partial G_i}{\partial \eta} + \Psi_{\eta}^{G_i}\right)(\eta_y)\right]. \quad (30)$$

La implementación de la CPML se lleva a cabo actualizando el arreglo de la variable Ψ para cada variable de flujo F y G a lo largo de su respectiva dirección ξ o η en cada iteración en el tiempo. En la dirección ξ, Ψ_x es obtenida como sigue:

$$\Psi_x^{n+1}(f) = b_x \Psi_x^{n}(f) + a_x(\partial_x f)^{(n+\frac{1}{2})}, \quad (31)$$

3.5 Esquema numérico

El esquema numérico sigue el modelo `predictor-corrector` de segundo orden en espacio y tiempo. Este método es iterativo explícito, en el paso *predictor* se aplica diferencias finitas hacia atrás para calcular una primera aproximación. En el paso corrector se aplica diferencias finitas hacia adelante para mejorar ese valor.

Aplicando el esquema a la ecuación (29) procede como sigue:
Paso predictor: La variable de flujo $U^{n+\frac{1}{2}}$ predicha es calculada como sigue:

$$U_{i,j}^{n+\frac{1}{2}} = U_{i,j}^{n} + \Delta t\left[\frac{1}{\kappa_{\xi_{i,j}}}\frac{F_{i-1,j}^{n} - F_{i,j}^{n}}{\Delta \xi_{i-1}} + \Psi_{\xi}^{n+\frac{1}{2}}[F^n]\right] + \frac{\Delta t}{\Delta \eta}(F_{i,j-1}^{n} - F_{i,j}^{n})\eta_{x_{i,j}} + \frac{\Delta t}{\Delta \eta}(G_{i,j-1}^{n} - G_{i,j}^{n})\eta_{y_{i,j}} + S_{i,j}^{n+\frac{1}{2}} \quad (32)$$

donde la viscosidad artificial $S_{i,j}^{n+\frac{1}{2}}$ es calculada como:

$$S_{i,j}^{n+\frac{1}{2}} = \frac{C_x|p_{i+1,j}^{n} - 2p_{i,j}^{n} + p_{i-1,j}^{n}|}{p_{i+1,j}^{n} + 2p_{i,j}^{n} + p_{i-1,j}^{n}} \times \left(U_{i+1,j}^{n} - 2U_{i,j}^{n} + U_{i-1,j}^{n}\right)$$

.

$$+\frac{C_y|p^n_{i,j+1}-2p^n_{i,j}+p^n_{i,j-1}|}{p^n_{i,j+1}+2p^n_{i,j}+p^n_{i,j-1}}\times\left(U^n_{i,j+1}-2U^n_{i,j}+U^n_{i,j-1}\right) \quad (33)$$

la viscosidad artificial es introducida para evitar posibles oscilaciones cuando se producen fuertes gradientes.

En la ecuación (39) C_x y C_y son dos parámetros que tienen valores típicos en un rango de 0.01 a 0.3. Su valor típico es determinado aproximadamente como $\text{máx}\left[(|U|+c)\Delta t\sqrt{\frac{1}{\Delta x^2}+\frac{1}{\Delta y^2}}\right]$, el cual es el orden de criterio de estabilidad CFL. Para esta aplicación se usan los valores $C_x = C_y = 0{,}1$.

De acuerdo con la formulación descrita en (Komatitsch and Martin, 2007), en el paso predictor la variable de evolución en el tiempo $\Psi[F^n]$ (variable de memoria) de la derivada de la variable de flujo F a lo largo de la dirección ξ puede ser calculada como:

$$\Psi_\xi^{n+\frac{1}{2}}[F^n_{i,j}] = b_x\Psi_\xi^{n-\frac{1}{2}}[F^n_{i,j}] + a_x\left[\frac{1}{\Delta\xi}\left(F^n_{i-1,j}-F^n_{i,j}\right)\right] \quad (34)$$

donde $b_x = e^{(d_x/k_x+\alpha_x)\Delta t}$ y $a_x = \frac{d_x}{k_x(d_x+k_x\alpha_x)(b_x-1).}$

La discretización de b_x y a_x procede como sigue:

$$b_{x,i} = e^{-(d_x/k_x+\alpha_x)\nu_i\Delta t} \text{ and } a_{x,i} = \frac{d_x}{k_x(d_x+k_x\alpha_x)}(b_{x,i-1}) \quad (35)$$

donde el subíndice i corresponde a la itésima iteración de un paso en el tiempo del esquema predictor-corrector. i toma los valores de 1 o 2 porque se trata de un esquema de segundo orden en tiempo y ν_i es una fracción del paso total en el tiempo que toma los valores de $\nu_1 = 0{,}5$ y $\nu_2 = 1{,}0$. Esto puede ser reformulado para órdenes más altos en tiempo (Martin et al., 2010). Para detalles mas extensos sobre la formulación de las funciones a_x y b_x se puede en consultar en (Komatitsch and Martin, 2007; Martin et al., 2008; Martin and Komatitsch, 2009; Zhang et al., 2009; Drossaert and Giannopoulos, 2007)

Antes de proceder con el paso corrector, es necesario decodificar la presión (p), de las variables de flujo U, como sigue:

$$p^{n+\frac{1}{2}}_{i,j} = (\lambda-1)\left[U^{n+\frac{1}{2}}_{4_{i,j}} - \frac{1}{2U^{n+\frac{1}{2}}_{1_{i,j}}}\left[\left(U^{n+\frac{1}{2}}_{2_{i,j}}\right)^2+\left(U^{n+\frac{1}{2}}_{3_{i,j}}\right)^2\right]\right] \quad (36)$$

con el valor de p es posible completar el paso predictor. Calculando las variables de flujo como: $F^{n+\frac{1}{2}}$ y $G^{n+\frac{1}{2}}$ como sigue: $F^{n+\frac{1}{2}}_{1_{i,j}} = U^{n+\frac{1}{2}}_{2_{i,j}}$, $F^{n+\frac{1}{2}}_{1_{i,j}} = \frac{\left(U^{n+\frac{1}{2}}_{2_{i,j}}\right)^2}{\left(U^{n+\frac{1}{2}}_{1_{i,j}}\right)^2} + p^{n+\frac{1}{2}}_{i,j}$,

.

$$F_{3_{i,j}}^{n+\frac{1}{2}} = \frac{U_{2_{i,j}}^{n+\frac{1}{2}} U_{3_{i,j}}^{n+\frac{1}{2}}}{U_{1_{i,j}}^{n+\frac{1}{2}}},\ F_{4_{i,j}}^{n+\frac{1}{2}} = \frac{U_{2_{i,j}}^{n+\frac{1}{2}}}{U_{1_{i,j}}^{n+\frac{1}{2}}}\left[U_{4_{i,j}}^{n+\frac{1}{2}} + p_{i,j}^{n+\frac{1}{2}}\right],\ G_{1_{i,j}}^{n+\frac{1}{2}} = U_{3_{i,j}}^{n+\frac{1}{2}},\ G_{2_{i,j}}^{n+\frac{1}{2}} = \frac{U_{2_{i,j}}^{n+\frac{1}{2}} U_{3_{i,j}}^{n+\frac{1}{2}}}{U_{1_{i,j}}^{n+\frac{1}{2}}},\ G_{3_{i,j}}^{n+\frac{1}{2}} = \frac{\left(U_{3_{i,j}}^{n+\frac{1}{2}}\right)^2}{\left(U_{1_{i,j}}^{n+\frac{1}{2}}\right)^2} + p_{i,j}^{n+\frac{1}{2}},\ G_{4_{i,j}}^{n+\frac{1}{2}} = \frac{U_{3_{i,j}}^{n+\frac{1}{2}}}{U_{1_{i,j}}^{n+\frac{1}{2}}}\left[U_{4_{i,j}}^{n+\frac{1}{2}} + p_{i,j}^{n+\frac{1}{2}}\right].$$

Paso Corrector:

La variable de evolución en el tiempo $\Psi[F^{n+\frac{1}{2}}]$ es calculada como:

$$\Psi_{\xi}^{n+1}\left[F_{i,j}^{n+\frac{1}{2}}\right] = b_x \Psi_{\xi}^{n-1}\left[F_{i,j}^{n+\frac{1}{2}}\right] + a_x\left[\frac{1}{\Delta\xi}\left(F_{i,j}^{n+\frac{1}{2}} - F_{i+1,j}^{n+\frac{1}{2}}\right)\right] \tag{37}$$

y finalmente la variable de flujo U en el siguiente tiempo $n+1$ es obtenida como:

$$U_{i,j}^{n+1} = U_{i,j}^{n} + \frac{1}{2}\left[U_{i,j}^{n+\frac{1}{2}} - U_{i,j}^{n} + \frac{1}{\kappa_{\xi_{i,j}}}\frac{F_{i,j}^{n+\frac{1}{2}} - F_{i+1,j}^{n+\frac{1}{2}}}{\Delta\xi_i} + \Psi_{\xi}^{n+1}\left[F_{i,j}^{n+\frac{1}{2}}\right] + \frac{F_{i,j}^{n+\frac{1}{2}} - F_{i,j+1}^{n+\frac{1}{2}}}{\Delta\eta}\eta_{x_{i,j}} + \right]\Delta t + S_{i,j}^{n+1}. \tag{38}$$

la viscosidad artificial $S_{i,j}^{n+1}$ es añadida en el paso corrector, como:

$$S_{i,j}^{n+1} = \frac{C_x|p_{i+1,j}^{n+\frac{1}{2}} - 2p_{i,j}^{n+\frac{1}{2}} + p_{i-1,j}^{n+\frac{1}{2}}|}{p_{i+1,j}^{n+\frac{1}{2}} + 2p_{i,j}^{n+\frac{1}{2}} + p_{i-1,j}^{n+\frac{1}{2}}} \times \left(U_{i+1,j}^{n+\frac{1}{2}} - 2U_{i,j}^{n+\frac{1}{2}} + U_{i-1,j}^{n+\frac{1}{2}}\right) + \frac{C_y|p_{i,j+1}^{n+\frac{1}{2}} - 2p_{i,j}^{n+\frac{1}{2}} + p_{i,j-1}^{n+\frac{1}{2}}|}{p_{i,j+1}^{n+\frac{1}{2}} + 2p_{i,j}^{n+\frac{1}{2}} + p_{i,j-1}^{n+\frac{1}{2}}} \times \left(U_{i,j+1}^{n+\frac{1}{2}} - 2U_{i,j}^{n+\frac{1}{2}} + U_{i,j-1}^{n+\frac{1}{2}}\right) \tag{39}$$

Para la descretización del tensor de esfuerzos viscosos o para los términos difusivos térmicos, se requiere una derivada con pesos, debido a la que la malla sobre el eje ξ no es uniforme.

Por simplicidad y evitar la escritura excesiva y repetitiva de formulaciones discretas de las derivadas de las componentes de velocidad y temperatura que se encuentran involucradas en los términos viscosos y los flujos termales, se denota a ϕ como una variable que puede tomar los valores de u, v and T. Las derivadas discretizadas de ϕ a lo largo de ξ y η son escritas como:

$$\left.\frac{\partial\phi}{\partial\xi}\right|_{i,j} = \frac{\phi_{i+1,j} + (\alpha_{\xi}^2 - 1)\phi_{i,j} - \alpha_{\xi}^2\phi_{i-1,j}}{\alpha_{\xi}(\alpha_{\xi}+1)\Delta\xi_i} \tag{40}$$

$$\left.\frac{\partial\phi}{\partial\eta}\right|_{i,j} = \frac{\phi_{i+1,j} + (\alpha_{\eta}^2 - 1)\phi_{i,j} - \alpha_{\eta}^2\phi_{i-1,j}}{\alpha_{\eta}(\alpha_{\eta}+1)\Delta\xi_i} \tag{41}$$

.

donde $\alpha_\xi = \frac{\Delta\xi_{i+1}}{\Delta\xi_i}$ y $\alpha_\eta = \frac{\Delta\eta_{j+1}}{\Delta\eta_j}$. Por lo tanto, se puede discretizar los esfuerzos viscosos y los términos difusivos térmicos de las ecuaciones (12), (13), (14), (15) y (16) utilizando las derivadas con peso (40) y (41), reemplazando ϕ por las componentes u y v o la temperatura T.

Capítulo 4

Diseño e implementación de la aplicación en OpenACC

Elaborar un buen diseño paralelo depende de muchos factores, desde conocer a profundidad el comportamiento de la aplicación para poder identificar dónde se concentra la mayor carga computacional, hasta tener un conocimiento más avanzado de cómo es que procesa la información en una GPU a nivel físico y lógico, como lo exige el diseño de programas con lenguajes como OpenCL o CUDA nativo.

En este capítulo se describe el diseño paralelo en OpenACC que resultó del análisis de la implementación previa en OpenMP de la descomposición de los ciclos que componen el código. Del mismo modo se especifica cada una de las directivas aplicadas a los ciclos y cómo afecta la declaración de variables al correcto comportamiento de los hilos que procesan los datos. Esta parte es relevante para la traducción del código al lenguaje OpenACC dado que el manejo de directivas y la estructura de los ciclos paralelos es muy similar al diseño en OpenMP.

4.1 Análisis del algoritmo

La principal acción para la adaptación del código serial es identificar dónde se encuentra la mayor carga de cómputo puesto que ahí es donde se sugiere reducir el tiempo de ejecución. En este caso es un método numérico en diferencias finitas que emplea un esquema *predictor-corrector* especificado en el capítulo anterior. Las características de este método hacen que el algoritmo sea paralelizable. El tiempo de ejecución que se emplea en la lectura de archivos, asignación de memoria o la inicialización de variables son despreciables porque se encuentran fuera del ciclo del tiempo y son solo ejecutados una vez durante todo el programa.

Por otro lado, los 16 ciclos que se encuentran dentro del ciclo principal denominado *paso en el tiempo* deben cumplir con ciertas características para poder paralelizarlos, por ejemplo: procesar los datos de forma SIMD, es decir, que no tengan dependencia entre iteraciones y además que sean computacionalmente intensivos.

La intensidad de un ciclo se refiere a *la relación entre las operaciones de punto flotante y los accesos a la memoria*. La forma de saber este parámetro es añadiendo

la bandera de compilación `-Minfo=intensity` para compiladores PGI Fortran. La bibliografía consultada sugiere que para un ciclo paralelizable la intensidad debe ser $I \geqslant 1$. La intensidad deseable es 4 pero si algún ciclo tiene una intensidad menor que 1 es posible paralelizarlo si forma parte de un programa más grande.

La figura 4.18 muestra los ciclos que componen el esquema numérico en el ciclo del tiempo. Son omitidos los primeros ciclos que corresponden a la inicialización de variables y a la construcción de las métricas. Después del código para calcular el volumen e inicializar la PML empieza el paso en el tiempo con un ciclo `while`, la primera parte correspondiente al paso *predictor* donde se calcula las variables de flujo con diferencias finitas hacia atrás y los ciclos van numerados del C_1 al C_7.

En la figura se muestran los ciclos del paso corrector numerados del C_8 al C_{13}; y los ciclos C_14 y C_{15} que calculan la energía por punto y la energía total respectivamente.

4.2 Vectorización de los ciclos

La vectorización es la ejecución de tipo SIMD en paralelo dentro de un solo núcleo de CPU. El núcleo realiza la misma operación de forma simultánea en N elementos adyacentes (vector). La eficiencia y el rendimiento dependen muchas veces de la estructura del código, sin embargo, es posible controlar la vectorización si se conoce como funciona tanto la aplicación cómo la vectorización y bajo qué condiciones opera.

El compilador de FORTRAN puede vectorizar los ciclos de manera automática, en la versión vectorial del eyector se empleó la auto-vectorización para aprovechar la estructura de bucles anidados. La vectorización de un ciclo va a depender de tres criterios para poder vectorizar adecuadamente. A continuación se describen.

SIN RAMIFICACIONES NI CICLOS INFINITOS. Los bucles que contienen instrucciones `EXIT` se descartan porque el contador debe estar definido, lo mismo ocurre cuando se discriminan elementos por medio de instrucciones `SWITCH` e `IF` porque inhiben la verctorización, al menos que la instrucción `IF` se aplique a todos los elementos aunque solo se almacene el resultado para los que cumplan con la condición. Todos los ciclos contenidos en el *paso del tiempo* cumplen con estas dos primeras condiciones.

REESTRUCTURACIÓN DE CICLOS ANIDADOS. En ocasiones el compilador cambia el orden de ejecución de ciclos anidados porque el recorrido es más accesible.

.

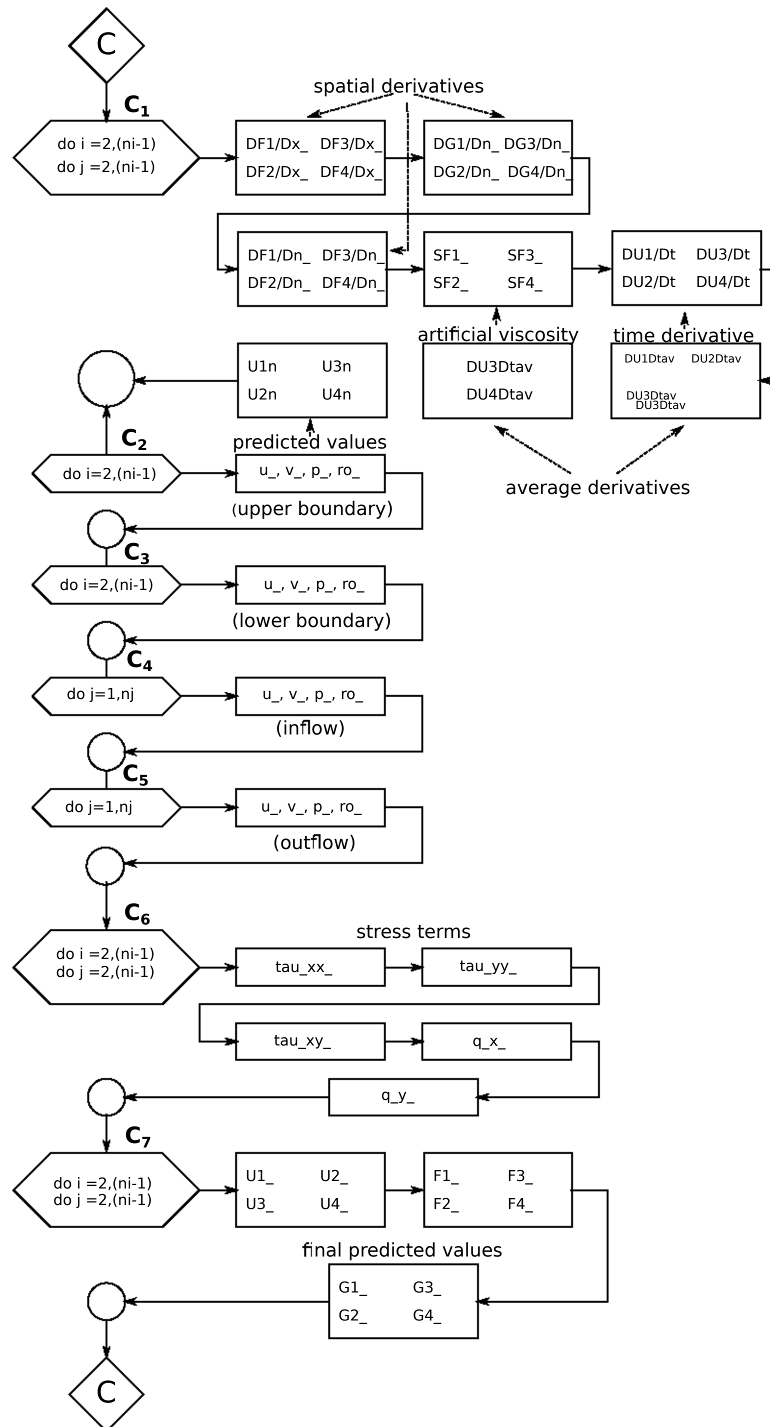


Figura 4.18: Diagrama de flujo del algoritmo numérico que emplea un esquema *predictor-corrector*

.

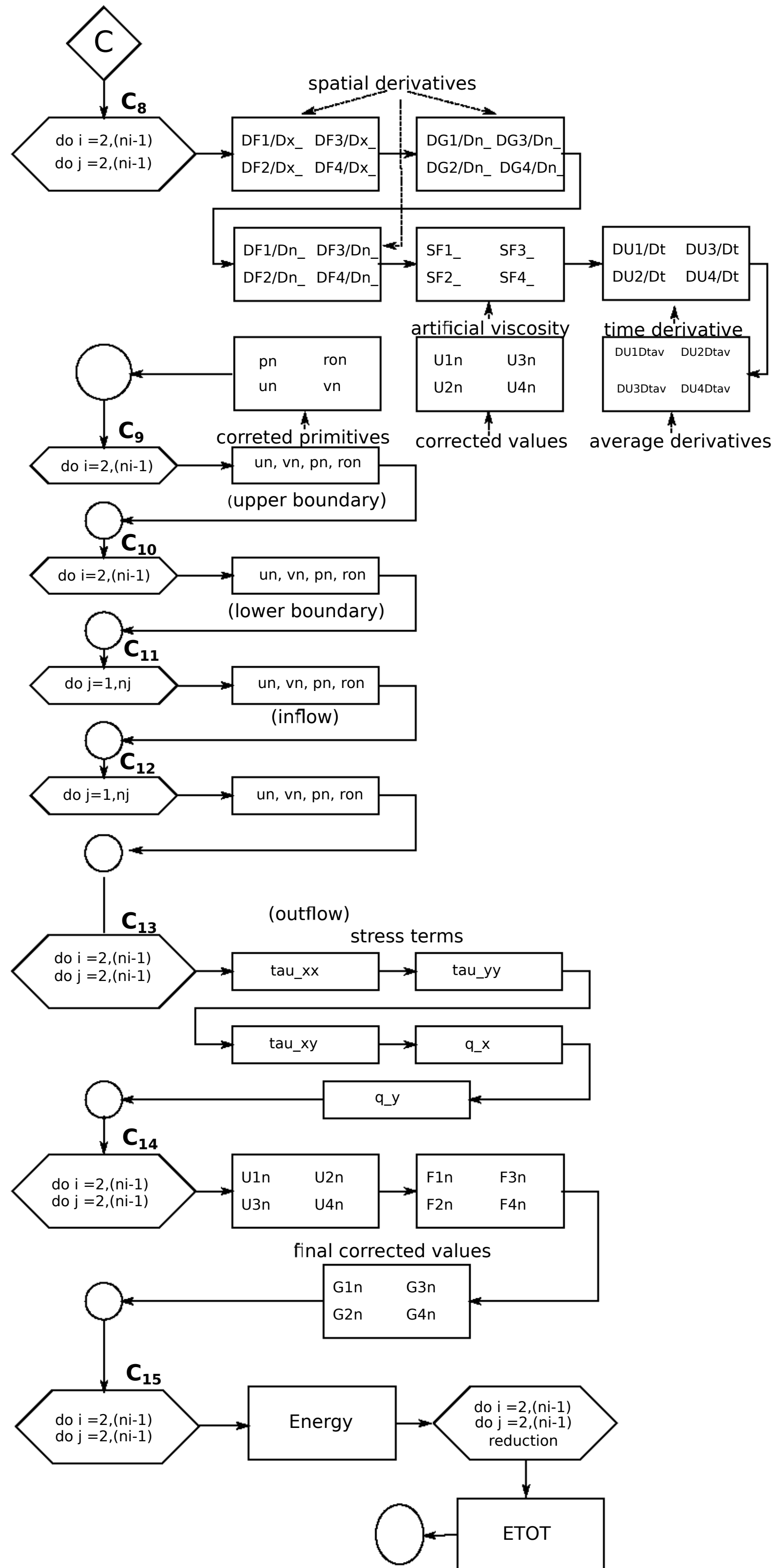


Figura 4.19: Diagrama de flujo del algoritmo numérico que emplea un esquema *predictor-corrector*
.

Cabe mencionar que la vectorización sólo se aplica al bucle interior.

Existen herramientas muy útiles que nos proporcionan información con respecto a las acciones que toma el compilador. Para el código vectorial del eyector se usó la bandera de compilación `-vect-report2` para generar un reporte de diagnóstico que muestra no solo si cambio el orden sino también las razones por las que pudo o no vectorizar y como ayuda adicional se ejecutó la opción `mbox-guide`, esta herramienta es muy útil pues sugiere formas para mejorar la vectorización, aunque en ocasiones aplicar las sugerencias puede ocasionar resultados adversos si no se conoce la aplicación, por lo que el buen funcionamiento depende del programador.

sin llamadas a funciones o instrucciones i/o. Se pueden vectorizar ciclos con llamadas a funciones si se identifican como *inlined* es posible vectorizar porque la función se inserta como parte del código principal.

4.3 Diseño en OpenMP

Desarrollar un código escrito en lenguaje OpenMP radica en el hecho de que es el estándar para acelerar código en arquitecturas de memoria compartida por lo que muchas aplicaciones están escritas utilizando las directivas. Además la creación de códigos paralelos es relativamente sencilla porque no es necesario rehacer el código serial.

En el diseño de OpenMP resaltan cuatro características que permitirán traducir el código con OpenMP a OpenACC pues los dos lenguajes se basan en directivas de compilación, solo requiere colocarlas en los puntos apropiados para crear ciclos paralelos o kernels en el caso de OpenACC. El diseño de OpenMP utilizado en esta investigación pretende generar un buen rendimiento a partir de la creación de una sola región paralela que englobe los ciclos contenidos en el paso del tiempo, con el fin de reducir la carga paralela que llevaría el crear y cerrar una región paralela por cada bucle que se pretenda acelerar.

Inicialmente, cuando se ejecuta el programa un solo hilo al que le llamaremos hilo maestro inicializa las variables; ese mismo hilo ejecuta el bucle que inicializa las variables primitivas u, v, ρ y P. Este bucle no fue paralelizado porque sólo se ejecuta una vez por lo que el tiempo de ejecución es despreciable. De igual manera el hilo maestro calcula M y T, introduce el flujo y recalcula M y T. El cálculo de las métricas de la transformación, así como los valores iniciales de los vectores de flujo y la introducción de las condiciones de frontera también son ejecutadas por

el hilo maestro.

Al llegar al constructor `!$OMP PARALLEL` se pone en marcha una serie de hilos esclavos, donde cada hilo tendrá una copia de las variables `k`, `id_t` y `num_t` que corresponden al contador de iteraciones, número de hilo y el número de hilos en ejecución, estas variables tiene la característica de no ser inicializadas antes de entrar a la región paralela y no se guarda su valor al salir de ella. Las variables declaras como `firstprivate` como `deltan`, `dentat` la energía total `ETOT` y el tiempo total `timetotal` son privadas a los hilos pero conservan el valor con el que se inicializaron antes de entrar a la región paralela. Las demás variables se declararon como `SHARED` de modo que se encuentran en la memoria global donde puede acceder cualquier hilo.

Para optimizar aún más los ciclos se crearon dos versiones del código OpenMP, la primera hacía uso de la directiva `!$OMP DO` la cual por sí sola sólo paraleliza el bucle exterior y ejecuta de forma serial las iteraciones del ciclo interior, para mayor optimización se agregó `COLLAPSE (2)` a los ciclos que tienen intensidad mayor a 1, cuando se colapsan los ciclos anidados todos los hilos que estén trabajando en la región paralela toman una iteración y la ejecutan, la distribución del trabajo va a depender del compilador porque no se especificó ningún `SCHEDULE`. Por otro lado, para manejar de manera eficiente los bucles que tiene una intensidad menor a la sugerida, como los ciclos que calculan las condiciones de frontera, se crearon cuatro tareas con la directiva `!$OMP TASK` donde la directiva `!$OMP single` limita la ejecución de las tareas a un solo hilo sin importar el orden de ejecución.

Después del paso corrector se calcula la matriz que almacena la energía en cada punto del dominio. Para calcular el acumulado de energía, es decir la variable `ETOT`, lo interesante es como opera la instrucción `COLLAPSE (2)` con la cláusula `REDUCTION`.

Para el cálculo del tiempo total todos los hilos tienen una copia de la variable `TimeTotal` por lo que es necesario especificar con la directiva que sólo un hilo va a imprimir en pantalla el tiempo total así como el número de iteración. Además, es aquí donde se genera el archivo para la visualización de la simulación en TECPLOT. Es hasta este punto donde dejan de operar los hilos que se crearon al inicio de la región paralela y finalmente el hilo maestro genera un reporte con los resultados de la ejecución.

.

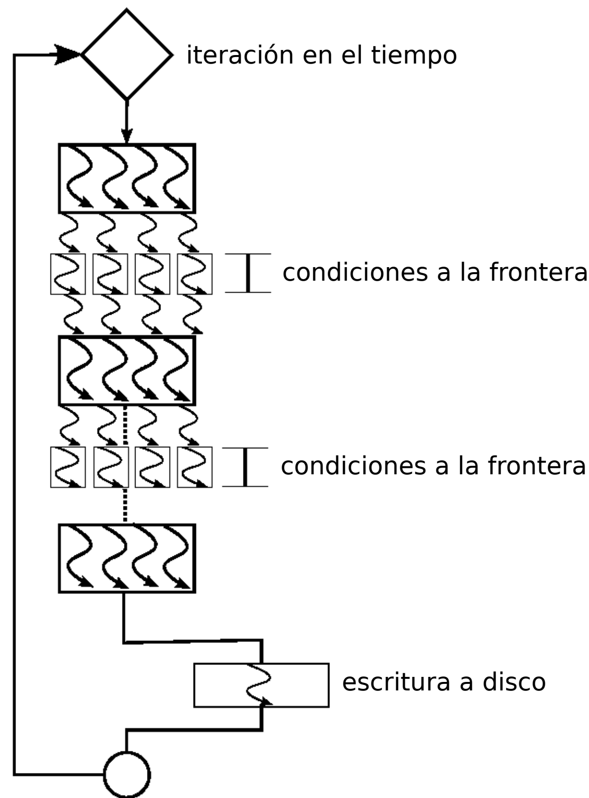


Figura 4.20: Diseño en OpenMP.

4.4 Diseño en OpenACC

Una vez adaptado el algoritmo al modelo OpenMP, es relativamente sencillo trasladarlo al modelo OpenACC. Las estrategias aplicadas en las investigaciones consultadas en el estado del arte sugieren que en un principio se considere mantener los datos el mayor tiempo posible en el dispositivo para reducir el tiempo de ejecución, y de esta forma mantener de persistente las variables en memoria, pues de ello depende reducir al mínimo la transferencia de datos entre el CPU y el GPU para lograr un buen rendimiento.

.

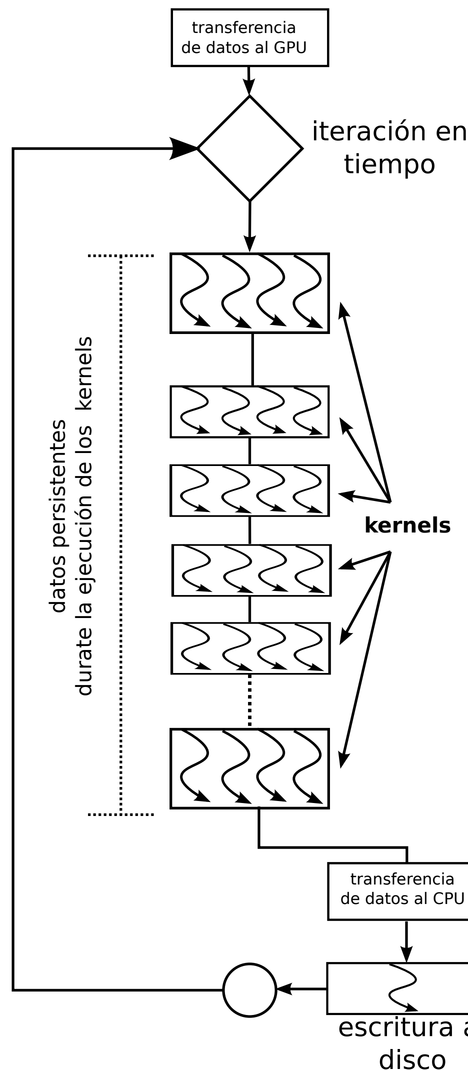


Figura 4.21: Modelo de aceleración en OpenACC

En la figura 4.21 se muestra el diagrama de ejecución del algoritmo de estudio modelado en OpenACC. El fragmento de código que corresponde a los ciclos donde se inicializan las variables de flujo y las primitivas se ejecutan en el CPU porque no contiene suficiente trabajo para explotar al máximo el paralelismo en el dispositivo. Antes de empezar el ciclo en el tiempo las variables junto con las instrucciones que conforman el modelo `predictor-corrector` son movidas a la memoria del dispositivo.

En OpenACC la directiva `kernel` le permite al programador especificar qué parte del código se ejecutará en el acelerador. La ventaja de esta directiva es que crea un *kernel* por cada ciclo contenido en la región, pero, por otro lado, la trasferencia

de variables al inicio y al cierre de cada región *kernel* repercute en el tiempo de ejecución, convirtiéndose en una desventaja. Este comportamiento es análogo a la apertura y cierre de regiones paralelas en OpenMP y para emular la solución aplicada en el modelo OpenMP se engloban la región kernel dentro de una región de datos representada con la directiva `!$acc data`, de esta forma los segmentos de cálculo intensivo como los ciclos `DO` utilizan las variables persistentes en la GPU y evitan la trasferencia excesiva de variables entre cada kernel.

Lo anterior se deriva en darle al dispositivo suficiente trabajo para explotar al máximo el paralelismo y por último una vez asignados los suficientes datos a procesar sugieren reutilizar los datos para evitar los cuellos de botella en el ancho de banda de la memoria.

La traducción del algoritmo paralelizado con OpenMP al modelo de aceleración OpenACC fue desarrollado bajo lo expuesto anteriormente. En una primera etapa se analizó el comportamiento de las variables en el algoritmo. Las variables se inicializaron antes de la región de datos, por lo que los valores ya se encuentran almacenados en la memoria local y es necesario copiarlos en la memoria del dispositivo. Siguiendo con el análisis, no es necesario regresar los valores finales a la memoria local, aunque sus valores hayan sido cambiados en el dispositivo, por lo que la clausula `copyin` es la mejor opción dado que al especificarse dentro de una región de datos, la memoria asignada a esas variables es liberada al finalizar la región, y así evitamos la trasferencia de datos.

Las variables restantes son calculadas dentro de la región de datos por lo que se le hace saber al compilador con la clausula `local` que los valores ya están presentes en la memoria del dispositivo durante la ejecución de los *kernels* dentro de la región de datos. El compilador interpreta que tiene que encontrar y usar los datos existentes en el acelerador. Es importante no olvidar declarar alguna de las variables que se calculan dentro de la región de datos porque el compilador interpreta que tiene que crearla y eso aumenta el tiempo de ejecución, porque se tendrían que crear y destruir cada vez que se ejecutara el *kernel* que hace uso de la variable creada.

El análisis de los ciclos `DO` estuvo dirigido al comportamiento de las directivas con respecto a los niveles de paralelismo con la estructura mostrada en el fragmento de Código 2.

En el fragmento de Código 2 se le indica al compilador que para cada ciclo debe generar el *kernel* correspondiente y que los ciclos internos al tener la clausula `independent` deben colapsarse. Cuando no se especifica el número de `gang` o `worker` el compilador lo decide en tiempo de ejecución; una vez asignados se

.

Código 2: Clausula PARALLEL

```
!$acc kernels
!$acc loop independent
DO j=2,(nj-1)
!$acc loop independent
        DO i=2,(ni-1)
$acc end kernels
```

Código 3: Actualización de la memoria en el host

```

IF( MOD(k,it_display) == 0 ) THEN
        print *,mod(k,it_display);
        !$acc update host(U,V,ro,P,T,M,ET)
        CALL LAYER_WRITE(DBLE(k))

END IF
```

mantienen fijos durante la ejecución. Por lo general las asignaciones del compilador son muy buenas para acelerar el código y por lo tanto no se dieron estos parámetros.

Por último, los valores de las variables que representan las primitivas y la energía total (u, v, ρ, P, T, M y ET) se actualizan en el host mediante la cláusula `update host` (ver fragmento de Código 3). La actualización es necesaria únicamente cuando se necesita escribir los valores a disco, ya que no existe una manera directa de que el dispositivo escriba los datos directamente a un dispositivo no volátil.

Capítulo 5

Pruebas de rendimiento y resultados de la simulación

En este capítulo se presentan las pruebas de rendimiento de las implementaciones en OpenMP ejecutadas tanto en CPU multi-core convencional. La implementación en OpenACC se ejecutó en una tarjeta NVIDIA TESLA C2070. La simulación numérica se realizó durante 3 segundos reales y, puesto que un paso en el tiempo Δt es de $1 \text{ x } 10^{-6}$, es necesario ejecutar tres millones de iteraciones para conseguir los 3 segundos de simulación real. El dominio computacional está compuesto de 1121×41 puntos discretos en los ejes ξ y η respectivamente.

Este problema está en el ámbito de los problemas que se consideran de fuerte escalamiento (*strong scaling*) por dos razones, la primera es porque su dominio es de tamaño constante e invariante en el tiempo y la segunda es porque el número de elementos de procesamiento es el que se incrementa (hilos). De acuerdo con lo anterior, el problema se rige por la ley de Amdahl. La ley de Amdahl mide la proporción de la ganancia del rendimiento obtenido, y depende de la fracción secuencial y paralela del programa, digamos que si duplicamos el número de procesadores, el tiempo de ejecución debería ser la mitad del tiempo que si se ejecutara en secuencial. Para el presente problema los procesos son hilos. Un programa se considera escalable linealmente si el tiempo de ejecución se reduce en $\frac{1}{n}$ para n elementos de procesamiento usados. En general llegar a una reducción lineal en el tiempo es improbable ya que la comunicación entre las unidades de procesamiento genera una sobrecarga.

5.1 Experimentos en CPU

La configuración de la estación trabajo utilizada es:

- CPU Dual Intel(R) Xeon(R) X5650 @ 2.67Ghz,
- 12 núcleos reales, 6 en cada socket, (un núcleo puede manejar hasta dos hilos cuando el Hyper-Threading (HT) está habilitado),
- 12 GB de memoria RAM,
- Sistema Operativo Cent OS 6.6,
- Compilador Intel Fortran 15.0.0.

Es necesario aclarar que aunque la versión del compilador que soporta OpenACC es PGI, se utilizó Intel Fortran 15.0.0 porque mostró un mejor rendimiento que el PGI para un procesamiento multi-hilo basado en CPU, por lo tanto, se ocupó el compilador que mejores resultados en tiempo generaba en el CPU.

Los experimentos para la implementación en OpenMP se llevaron a cabo teniendo en cuenta los efectos de la vectorización y el HT en el rendimiento. Los experimentos se ejecutaron con el HT habilitado y deshabilitado. Para realizar los experimentos se crearon de 1 a n hilos, siendo n el número de núcleos. No es necesario crear más hilos que núcleos (físicos o lógicos) en el sistema porque para un problema en *strong-scaling* se crearía sobrecarga. Para analizar el comportamiento del rendimiento cuando el HT está desactivado se crearon de uno a 12 hilos por ser el número de núcleos físicos. Por otro lado, cuando el HT está habilitado se crean de uno a 24 hilos kernel.

En los experimentos con el HT desactivado, la estrategia es asignar un hilo a un núcleo, y mantener todos los hilos corriendo sobre un único procesador físico asignado uno a uno; a esto se le llama distribución compacta (ver figura 5.22). Los experimentos de rendimiento para las 30,000 iteraciones muestran que la asignación por afinidad dispersa es un poco más lenta que la afinidad compacta de un 30 % a 10 % (desde dos a 12 hilos).

Cuando el HT está habilitado, cada núcleo de procesamiento puede manejar dos hilos de procesamiento por lo que se crea un núcleo virtual más dando un total de 24 núcleos virtuales. El comportamiento que se genera en la afinidad dispersa y compacta cuando el HT está activado se muestra en la Figura 5.23.

Para llevar a cabo los experimentos de rendimiento se tomó en cuenta los efectos de la vectorización de los ciclos internos y en otros la operación `collapse` (colapsamiento de ciclos), que consiste en incrementar el número total de iteraciones que serán distribuidas entre los hilos llevando a cabo el producto cruz entre el número de iteraciones en el ciclo exterior contra el número de iteraciones en el ciclo interior. Es posible que cuando los procesadores trabajan con gran cantidad de operaciones aritméticas el rendimiento pueda aumentar al aplicar el colapsamiento de ciclos, porque aumenta el número total de iteraciones que son distribuidas entre los hilos disponibles disminuyendo la intensidad computacional. Para esta aplicación la vectorización de los ciclos internos resultó tener mejor rendimiento que los ciclos colapsados de un 5 % a 7 %. Cuando se vectoriza un algoritmo lo que hacemos es ejecutar una operación sobre un grupo de valores contiguos, todo al mismo tiempo. Para este caso la vectorización sólo se aplica a ciclos internos que corresponden a operaciones de punto flotante. Es necesario comentar que la vectorización y el colapsamiento no se pueden aplicar al mismo tiempo porque

.

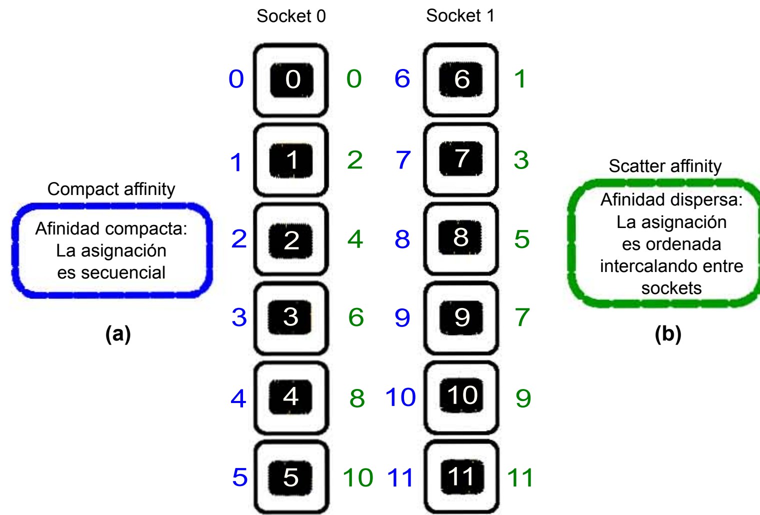


Figura 5.22: Comportamiento de afinidades en un SMP con dos Sockets y seis núcleos por socket, con HT desactivado; (a) asignación compacta, (b) asignación por dispersión.

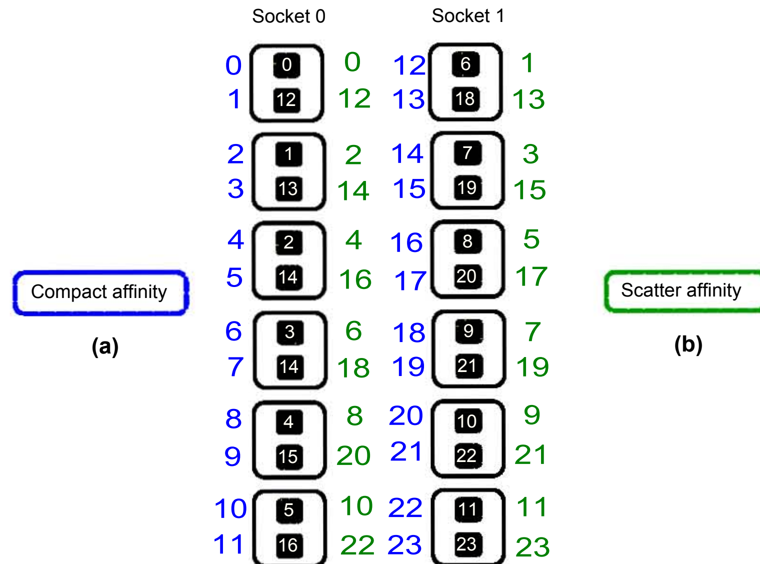


Figura 5.23: Comportamiento de afinidades en un SMP con dos sockets y seis núcleos por socket, con HT habilitado; (a) afinidad compacta (b) afinidad dispersa. Los rectángulos significa que los núcleos virtuales son manejados por el mismo núcleo físico.

cuando los ciclos se colapsan: la granulidad disminuye y la vectorización no podría aplicarse, es decir, por el colapso de los ciclos el cómputo es insuficiente para la vectorización.

Para optimizar el cálculo de las condiciones de frontera se crearon tareas con la directiva TASK de OpenMP. Los resultados arrojaron que la ganancia de rendimiento fue de 1 % a 1.5 % comparado con ciclos paralelizados convencionalmente; sin embargo, la ganancia es baja. También se mantuvo una región paralela persistente de manera que le restaran tiempo de latencia al abrir y cerrar las regiones; la ganancia de esta estrategia es de 0.5 % y 1.0 %. Por lo que se concluye que no hay sobrecarga notable cuando se utilizan muchas regiones paralelas, de manera que se pueden usar una o varias regiones paralelas.

La Tabla 1 muestra el tiempo de cómputo con su respectivo *speed-up* (factor de rendimiento) con el HT desactivado y en la Tabla 2 con el HT activado. El mejor rendimiento se registró al considerar la asignación compacta, la vectorización de ciclos internos, creación de tareas para condiciones de frontera y el HT desactivado. Como el rendimiento entre la afinidad compacta y dispersa es similar para el caso del HT activado, se muestra únicamente el resultado con la afinidad compacta ya que tiene un rendimiento un poco mejor que la dispersa. Cuando el HT esta activado podemos observar una discrepancia debido a que la afinidad compacta asigna dos hilos al mismo núcleo físico por lo cual lleva a su máximo procesamiento el núcleo y es muy similar su comportamiento a cuando el HT está desactivado, la afinidad dispersa como es de esperarse en un principio proporciona un mejor rendimiento debido a que un hilo es asignado solamente a un núcleo físico. No obstante, cuando se alcanza el máximo número de núcleos físicos, el rendimiento empieza a decaer y se empalma con la afinidad compacta.

La Figura 5.24 muestra el correspondiente comportamiento del tiempo de cálculo con el HT desactivado (inciso (a)) y activado (inciso (b)) y se observa que cuando el HT está habilitado la ejecución en serie pierde en un 20 % el rendimiento. Este resultado corresponde a los obtenidos en la investigación de (Zhang et al., 2004), quienes encontraron que a veces es mejor ejecutar una aplicación OpenMP utilizando únicamente un solo hilo por procesador físico. No todas las aplicaciones HPC ganan rendimiento con el HT activado, Ali and Leng's (2002) mostraron en sus estudios que cuando HT se utiliza para códigos de cálculo intensivo de punto flotante, que necesitan compartir las variables entre los hilos, se puede generar una sobrecarga. Para lograr un rendimiento similar con un solo hilo sin el HT, se necesita la creación de dos hilos y asignarlos al mismo núcleo. Por el contrario, para aplicaciones en las que los hilos no comparten el dominio computacional se ha encontrado una ganancia en el rendimiento acorde con (Couder-Castañeda et al., 2015).

El gráfico de la Figura 5.24(a) muestra un comportamiento gobernado por la ley de Amdahl con un factor máximo de aceleración de 6.14 X. Para determinar la sobrecarga existente es decir, la fracción serial se grafica la ley de Amdahl con

.

Tabla 1: Resultados de tiempo de cómputo y *speed-up* obtenidos con OpenMP sobre Xeon-CPU con HT desactivado y afinidad compacta. Los factores de *speed-up* se calcularon con la mejor version serial optimizada como referencia. Con HT-Hyper Threading, V-VectorizaciÓn, D-Desactivado and A-Activado

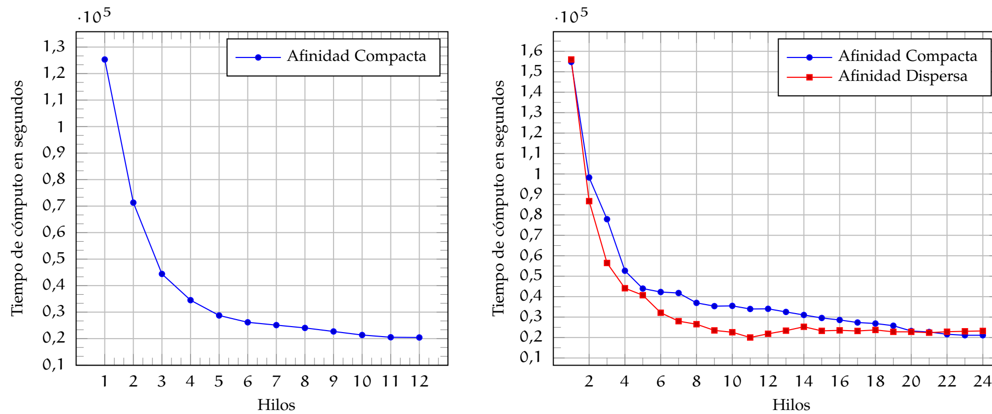
Hilos	HT-D V-A	*Speed-up*
1	34h48m32s	(1.00X)
2	19h26m58s	(1.79X)
3	12h20m02s	(2.82X)
4	09h35m11s	(3.63X)
5	07h58m53s	(4.36X)
6	07h15m54s	(4.79X)
7	06h58m14s	(4.99X)
8	06h40m30s	(5.21X)
9	06h18m21s	(5.52X)
10	05h55m56s	(5.87X))
11	05h41m52s	(6.11X)
12	05h40m23s	(6.14X)

Tabla 2: Tiempo de cómputo con su respectivo *speed-up* obtenido con HT habilitado para afinidad compacta y afinidad por dispersión. HT-Hyper Threading, V-Vectorización, A-Activo.

Hilos	HT-A,V-A,A-C	HT-A V-A,A-D	Hilos	HT-A V-A,A-C	HT-A V-A,A-D
1	43h22m22s(1.00X)	43h19m58s(1.00X)	13	09h01m54s(4.8X)	06h29m24s(6.68X)
2	27h17m46s(1.59X)	24h05m16s(1.80X)	14	08h37m13s(5.03X)	07h01m00s(6.18X)
3	21h38m07(2.00X)	15h41m05(2.76X)	15	08h12m43s(5.28X)	06h27m47s(6.70X)
4	14h37m51s(2.96X)	12h15m17s(3.54X)	16	07h56m28s(5.46X)	06h31m54s(6.63X)
5	12h12m27s(3.55X)	11h18m27s(3.83X)	17	07h35m52s(5.71X)	06h26m55s(6.72X)
6	11h44m25s(3.69X)	08h55m26s(4.86X)	18	07h27m38s(5.81X)	06h35m04s(6.58X)
7	11h36m13s(3.74X)	07h46m44s(5.57X	19	07h09m47s(6.06X)	06h20m00s(6.84X)
8	10h15m49s(4.23X)	07h21m59s(5.88X)	20	06h26m38s(6.73X)	06h19m51s(6.84X)
9	09h49m17s(4.42X)	06h31m26s(6.64X)	21	06h18m32s(6.87X)	06h12m58s(6.97X)
10	9h51m28s(4.40X)	06h16m28s(6.91X)	22	06h00m01s(7.23X)	06h20m40s(6.83X)
11	09h26m09s(4.60X)	05h33m41s(7.79X)	23	05h51m48s(7.40X)	06h24m12s(6.77X)
12	09h27m33s(4.59X)	06h03m58s(7.14X)	24	05h51m49s(7.40X)	06h27m04s(6.72X)

$f = 0{,}05$ y $f = 0{,}10$ y de lo que se observa es que la fracción serial del código se

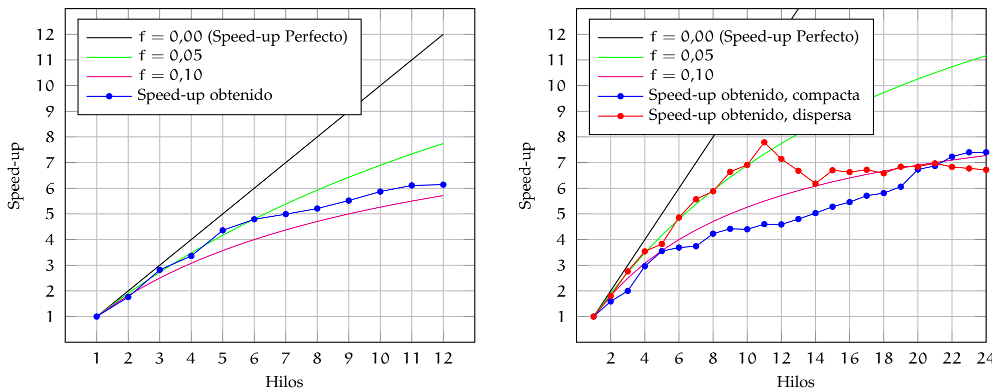
.



(a) Tiempo de cómputo obtenido con HT deshabilitado (12 hilos)

(b) Tiempo de cómputo obtenido con HT habilitado (24 hilos)

Figura 5.24: Tiempo de computo obtenido en un CPU Xeon Dual.



(a) El *Speed-up* se obtuvo usando el tiempo de computo mostrado en la Figura 5.24(a).

(b) El *Speed-up* se obtuvo usando el tiempo de cómputo mostrado en la Figura 5.24(b).

Figura 5.25: Comparación de los factores de velocidad contra la ley de Amdahl con una fracción serial de $f = 0{,}05$ y $f = 0{,}10$. En el inciso (a) se muestran un comportamiento gobernado por la ley de Amdahl, en el inciso (b), se muestra como la afinidad en un principio tiene una mayor rendimiento, esto es debido a que se utiliza un núcleo físico, pero cuando cuando se empieza a procesar dos hilos por núcleo físico, el rendimiento es muy similar.

encuentra entre dichos valores, por lo que la sobrecarga no supera el 10 %.

Como ya se había mencionado, cuando se activa el HT cada núcleo puede manejar dos hilos y reportar 24 núcleos lógicos. La estrategia es similar a la anterior, se crearon de 1 a 24 hilos para analizar el comportamiento del rendimiento con el fin de probar que se puede mejorar el rendimiento para esta aplicación con el

.

HT activado. Los resultados para esta prueba mostraron que el mejor factor de aceleración obtenido es de 6.72 X para la afinidad dispersa y 7.40 X para la afinidad compacta, considerando como referencia el tiempo de cómputo de la versión serial optimizada con HT activado. Para este caso de estudio, el uso de HT no aumenta el rendimiento, por lo tanto el HT en realidad no proporciona un beneficio para esta aplicación. Este comportamiento parece ser normal, porque la aplicación es de punto flotante intensivo y dos hilos están compartiendo la misma unidad de punto flotante (FPU) del núcleo.

5.2 Experimentos en la GPU

Los experimentos en GPU se llevaron a cabo en una tarjeta Tesla C2070 con las siguientes características:

- 14 microprocesadores (SM),
- 32 núcleos por multiprocesador (total de 448 núcleos GPU),
- 6 GB de memoria global DDR5,
- Hasta 515 Gflops teóricos en punto flotante de doble precisión,
- Hasta un Teraflop teórico en punto flotante de simple precisión,
- 1.15 GHz de frecuencia en los núcleos de la GPU.

Para ejecutar los experimentos se utilizó la versión 15,3 del compilador PGI para 64 bits con las banderas de compilación: `—Minline` `—Minfo=acc`, `accel`, `intensity` `—acc` `—fast`. La especificación del uso de cada una de las banderas se describe en el capítulo dos del presente trabajo.

El tiempo de ejecución fue de 3 horas 31 minutos 47 segundos, el cual resulto mejor que el tiempo de ejecución en la arquitectura Dual Xeon con 12 núcleos. La tabla 3 compara los factores de rendimiento de la tarjeta C2070 con respecto al CPU en su mejor versión serial y paralela.

5.3 Validación del código traducido

Como se ha descrito, el presente trabajo emplea dos arquitecturas diferentes (CPU convencional, GPU NVIDIA) y dos estándares de programación (OpenMP y OpenACC) para obtener el máximo rendimiento de cada plataforma. Es normal que al traducir un algoritmo de un paradigma a otro se cometan errores lógicos que afecten los resultados de la simulación. Para reducir los errores inherentes

Tabla 3: Comparación del factor de velocidad tomando como referencia la GPU.

GPU	Tiempo de cómputo	vs mejor version serial en el CPU	vs mejor versión paralela en el CPU
C2070	03h31m47s (1.0 X)	34h48m32s (9.86 X)	5h 48m 23s (1.6 X)

durante el desarrollo se busca que las gráficas de temperatura y/o gráficas de alguna variable significativa sean las mismas para todas las versiones de código.

El flujo supersónico de un eyector es un problema muy complejo debido a la física del flujo y de la geometría del eyector. Es necesaria una transformación de coordenadas para adaptar la sección de expansión del difusor al plano computacional discreto, mostrado en el capítulo 3 del presente trabajo. Por otro lado, como el flujo es supersónico, las ondas de presión están presentes en el flujo y tienen que ser absorbidas en la frontera de salida, evitando que la simulación muestre un rebote de flujo ficticio. Por esta razón, se implementaron condiciones de frontera PML convolucional o CPML para absorber las ondas de presión (Martin and Couder-Castaneda, 2010a). Además, el esquema numérico considera una simulación numérica directa, lo que implica alto costo computacional incluso a bajos números de Reynold. Esta es una de las razones por las que un algoritmo con alto nivel de optimización debe ser utilizado para este problema.

Una de las zonas de cuidado es aquella que fue modificada para adaptarla al dominio computacional, por ello se almacenaron los valores de las variables primitivas para analizar la calidad de los cálculos. Por esta razón, un visor numérico se encuentra cerca de una de las paredes inclinadas (ver Figura 5.26).

Otro aspecto importante con respecto a la ubicación del visor es que a causa del flujo espurio proveniente de las fronteras abiertas podrían generarse distorsiones de la estructura de flujo, por lo que el monitoreo de esta zona es relevante y se considera una región crítica. La generación de puntos de datos por variable primitiva de flujo se almacena cada 30,000 iteraciones de 3 millones, generando 100 puntos. En la figura 5.26 se observa una zona de 10 cm donde se aplica la CPML en la frontera abierta para absorber las ondas de presión.

Otro aspecto relevante como identificador de una buena adaptación de un código es que el cálculo de todas las variables involucradas en el algoritmo coincida con la versión serial o de referencia. Para el caso de estudio la simulación numérica será evaluada comparando el procesamiento en serie en un solo núcleo y la solución paralela de igual manera ejecutada en un solo núcleo. Para fines prácti-

.

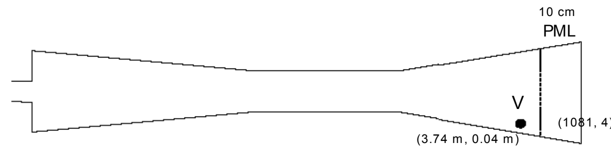


Figura 5.26: Ubicación del visor numérico cerca de la zona de salida. El espesor de la zona de absorción CPML es de 10 cm ($10 \times \Delta x$).

cos se analiza el cálculo de la energía total del sistema (ET), ya que se conforma de todas las variables primitivas, como se muestra a continuación.

$$E_t = \frac{1}{2}(u^2 + v^2) + \frac{1}{\gamma - 1}\frac{p}{\rho}, \tag{42}$$

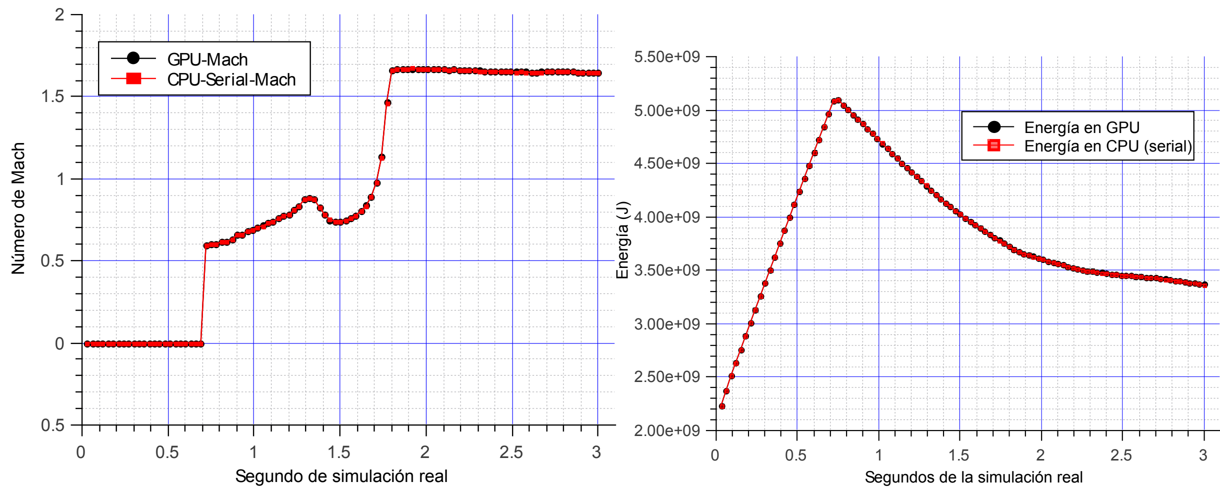
donde u y v son los vectores con los componentes de velocidad en la dirección x y y respectivamente, p es la presión, ρ es la densidad el flujo y γ es el coeficiente de dilatación adiabática. El primer término del lado derecho de la ecuación (42) representa la energía cinética por unidad de masa y el segundo término representa la energía interna por unidad de masa.

Otra variable a considerar es el número de Mach (M), recordemos que esta variable muestra la relación de la velocidad del objeto de estudio con respecto a la velocidad del sonido, por lo que se vuelve un parámetro a comparar por caracteriza al fluido bajo ciertas consideraciones.

Con fines de entendimiento, sólo se expone el resultado que alcanzó el mejor rendimiento en el CPU (ver figura 5.27). En las figuras antes mencionadas podemos observar que las curvas de ambos resultados así como el coeficiente de correlación muestran una buena relación entre la solución serial y la solución paralela traducida. Sin embargo se observaron pequeñas variaciones entre los resultados de la plataforma. Incluso dentro de las mismas plataformas donde se incrementan los hilos de procesamiento.

Dicho lo anterior, el algoritmo numérico debe ser lo suficientemente robusto y estable durante largos periodos con el fin de demostrar la estabilidad numérica. Lo que se muestra en la Tabla 4 es la diferencia de las magnitudes de la energía total (ET) y el número de Mach monitoreado en el visor V al final de los 3 seg de simulación, para el CPU y GPU. Los correspondientes porcentajes de error relativo se muestran en la Tabla 5 en relación con la solución serial de referencia. Se puede observar que la arquitectura de la CPU multi-núcleo ofrece la solución más precisa con respecto a la solución serial de referencia, donde el error es despreciable, tanto para la energía total como para el número de Mach.

.



(a) Número de Mach, Coef. de correlación = 0,999999483128566.

(b) Energía total, Coef. de correlación = 0,999998594006821.

Figura 5.27: Comparación de la energía total y el número de Mach correspondiente al visor *v*. Simulaciones en GPU comparado con la solución serial de referencia

Tabla 4: Energía total y número de Mach obtenido en el visor V para las CPU y GPU al final de la simulación de 3 seg.

	E_t (m^2/s^2)	M
Referencia	3366417302,615026	1,644826
GPU	3366759849,759678	1,644971
CPU multi-core	3366417302,615055	1,644826

Tabla 5: El porcentaje de error relativo para la energía total y el número de Mach obtenido en el visor V del GPU comparado contra la versión serial en el CPU.

	Error para E_t (%)	Error para M (%)
GPU	$1{,}01754 \times 10^{-4}$	$8{,}81552 \times 10^{-5}$

Es importante señalar que el flujo alcanza su forma estacionaria a los tres segundos de simulación y por tal motivo las pruebas de rendimiento se hicieron sobre este tiempo. No obstante, para obtener una secuencia de imágenes en forma de película se simuló hasta 10 segundos. La Figuras 5.28, 5.29 y 5.30 muestran diferentes capturas del número de Mach, densidad y temperatura respectivamente, a lo largo del difusor en cinco diferentes momentos de la simulación: $0,08$ segundos, $2,0$ segundos, $4,0$ segundos, $8,0$ segundos y $10,0$ segundos para cada captura. Los patrones se obtuvieron en la GPU C2070. El proceso por el cual se rige un eyector

para procesar el fluido son la compresión, transferencia y expansión en el difusor, lo cual puede ser observado en las gráficas.

.

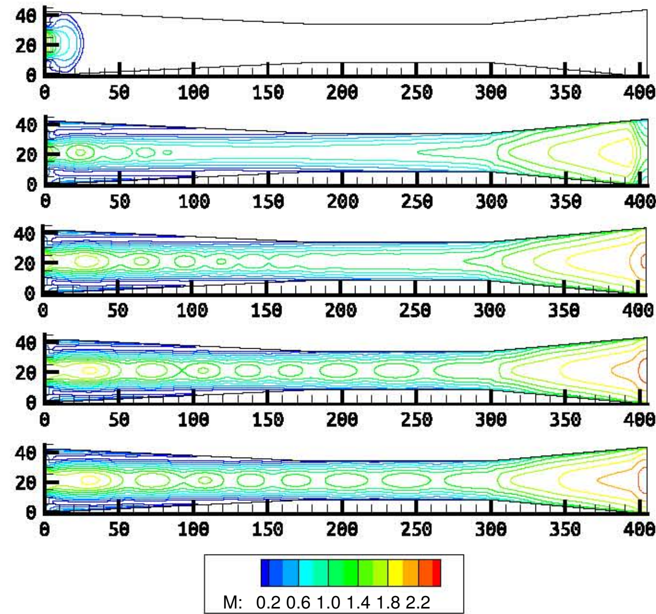


Figura 5.28: Gráficas de contorno del número de Mach (M) obtenido en diferentes tiempos de la simulación a 80 mil, 2 millones, 4 millones, 8 millones y a 10 millones de iteraciones de arriba hacia abajo.

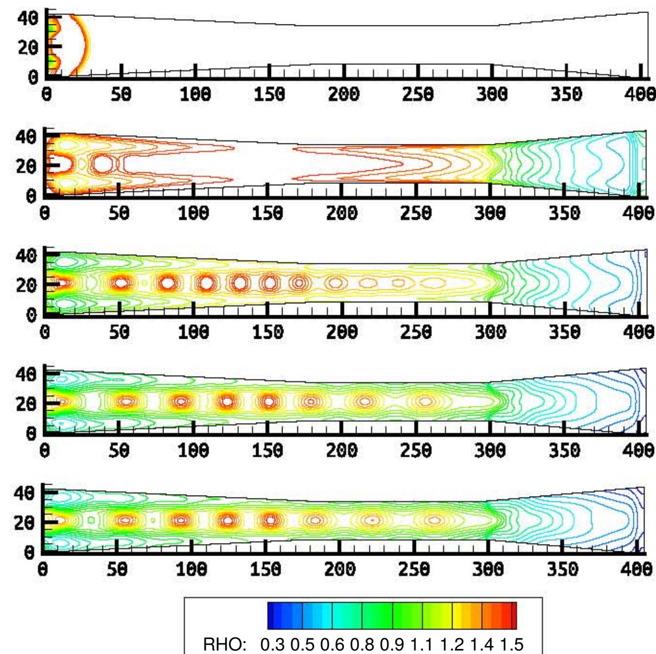


Figura 5.29: Gráficas de contorno de la densidad (ρ) obtenido en diferentes tiempos de la simulación a 80 mil, 2 millones, 4 millones, 8 millones y a 10 millones de iteraciones de arriba hacia abajo

.

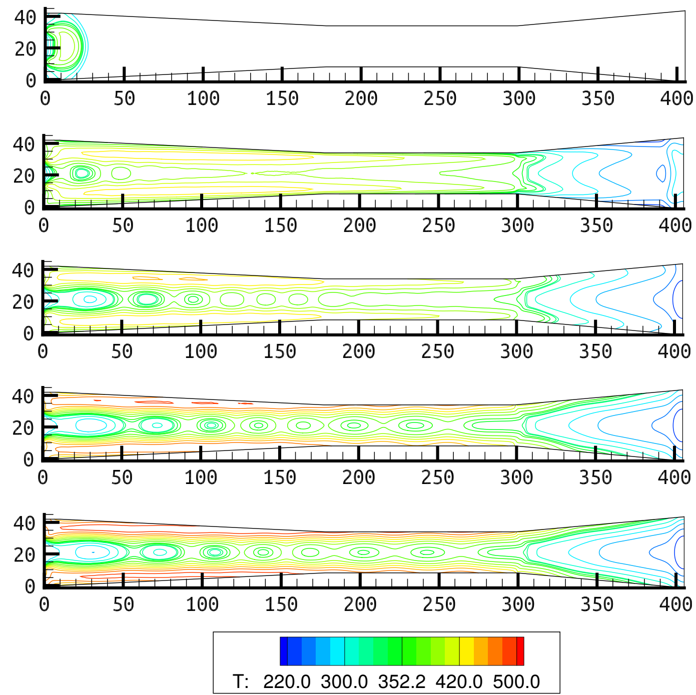


Figura 5.30: Gráficas de contorno de la temperatura (T) obtenida en diferentes tiempos de la simulación a 80 mil, 2 millones, 4 millones, 8 millones y a 10 millones de iteraciones de arriba hacia abajo.

Conclusiones

La utilización de las tarjetas NVIDIA surgió de la necesidad de reducir el tiempo de cómputo de las aplicaciones de ingeniería. No obstante, debido a que son una arquitectura distinta a la del GPU, el modelo de programación tiene que cambiar. En un principio, las aplicaciones científicas se empezaron a migrar a tarjetas gráficas utilizando CUDA C, lo cual elevaba el tiempo de programación y de depuración de las aplicaciones, por tal motivo, surgió OpenACC con la finalidad de reducir el tiempo de programación basada en una metología basada en directivas similar a lo que significa OpenMP para los sistemas multi-núcleo.

La aplicación adaptada en esta tesis originalmente era un código serial para un flujo supersónico. Los resultados obtenidos muestran que la versión OpenMP reduce el tiempo de cálculo en un sistema multi-núcleo de acuerdo con la ley de Amdhal. Por otro lado la estrategia desarrollada en OpenACC que propone mantener las regiones de datos activas mientras los kernels son ejecutados, resultó ser la mejor, pues evita en gran medida la transferencia de datos entre el CPU y la GPU y manteniendo las variables persistentes cuando se trabaja. Por lo que se mejora el rendimiento de la aplicación sin invertir tanto esfuerzo y tiempo de desarrollo. De hecho el GPU C2070 resultó ser 1.6 X más rápido que los 12 núcleos integrados en la estación de trabajo y considerando que el consumo energético y de adquisición de una GPU es mucho menos onerosa que el de una estación de trabajo, la GPU se convierte en un excelente hardware para llevar a cabo simulaciones numéricas.

Como conclusión general, de acuerdo con los resultados de programación y productividad muestran que los estándares basados en directivas como OpenMP y OpenACC son buenas alternativas para acelerar la ejecución del código, dado que el rendimiento obtenido con ambas es satisfactorio y que a partir de un código OpenMP sí se tiene una buena estrategia es posible migrarlo a OpenACC sin mucho esfuerzo de programación.

Considerando que OpenMP y OpenACC son paradigmas que expresan el paralelismo, es decir, que el realmente encargado de generar el código paralelo es el compilador, el rendimiento obtenido es muy aceptable considerando que implica menos tiempo de desarrollo comparado con desarrollar kernels CUDA nativos.

Bibliografía

Ali, R. and Leng, T. (2002). An empirical study of hyper-threading in high performance computing clusters, *Super computing 2002*.

Amritkar, A., Tafti, D., Liu, R., Kufrin, R. and Chapman, B. (2012). Openmp parallelism for fluid and fluid-particulate systems, *Parallel Computing* .

Anderson, J. D. and Wendt, J. (1995). *Computational fluid dynamics*, Vol. 206, Springer.

Bednarz, T., Domanski, L. and Taylor, J. (2011). Computational fluid dynamics using opencl - a practical introduction.

Beyer, J., Stotzer, E., Hart, A. and De Supinski, B. (2011). Openmp for accelerators, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* .

Cloutier, B., Muite, B. and Rigge, P. (n.d.). Performance of fortran and c gpu extensions for a benchmark suite of fourier pseudospectral algorithms, *2012 Symposium on Application Accelerators in High Performance Computing*.

Couder-Castaneda, C. (2009). Simulation of supersonic flow in an ejector diffuser using the jpvm, *Journal of Applied Mathematics* **2009**.

Couder-Castañeda, C., Ortiz-Alemán, J. C., del Castillo, M. G. O. and Nava-Flores, M. (2015). Forward modeling of gravitational fields on hybrid multi-threaded cluster, *Geofísica Internacional* **54**(1): 31 – 48.

Drossaert, F. H. and Giannopoulos, A. (2007). A nonsplit complex frequency-shifted pml based on recursive integration for fdtd modeling of elastic waves, *Geophysics* **72**(2): T9–T17.

Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G. and Dongarra, J. b. (2012). From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming, *Parallel Computing* **38**(8): 391–407. cited By (since 1996) 5.

Foster, I. (1995). Designing and building parallel programs.

Han, T. D. and Abdelrahman, T. S. (2011). hicuda: High-level gpgpu programming, *Parallel and Distributed Systems, IEEE Transactions on* **22**(1): 78–90.

Komatitsch, D. and Martin, R. (2007). An unsplit convolutional Perfectly Matched Layer improved at grazing incidence for the seismic wave equation, *Geophysics* **72**(5): SM155–SM167.

Lee, S., Min, S.-J. and Eigenmann, R. (2009). Openmp to gpgpu: a compiler framework for automatic translation and optimization, *ACM Sigplan Notices* **44**(4): 101–110.

Martin, R. and Couder-Castaneda, C. (2010a). An improved unsplit and convolutional perfectly matched layer absorbing technique for the navier-stokes equations using cut-off frequency shift, *CMES - Computer Modeling in Engineering and Sciences* **63**(1): 47–77.

Martin, R. and Couder-Castaneda, C. (2010b). An improved unsplit and convolutional perfectly matched layer absorbing technique for the navier-stokes equations using cut-off frequency shift, *Computer Modeling in Engineering & Sciences(CMES)* **63**(1): 47–77.

Martin, R. and Komatitsch, D. (2009). An unsplit convolutional perfectly matched layer technique improved at grazing incidence for the viscoelastic wave equation, *Geophys. J. Int.* **179**(1): 333–344.

Martin, R., Komatitsch, D. and Gedney, S. D. (2008). A variational formulation of a stabilized unsplit convolutional perfectly matched layer for the isotropic or anisotropic seismic wave equation, *Comput. Model. Eng. Sci.* **37**(3): 274–304.

Martin, R., Komatitsch, D., Gedney, S. D. and Bruthiaux, E. (2010). A high-order time and space formulation of the unsplit perfectly matched layer for the seismic wave equation using Auxiliary Differential Equations (ADE-PML), *Comput. Model. Eng. Sci.* **56**(1): 17–42.

Mott, R. L. (1996). *Mecánica de fluidos aplicada*, Pearson Educación.

NVIDIA, k. (2014). Nvidia's next generation cuda compute architecture, *NVIDIA, Santa Clara, Calif, USA* .

Reyes, R., López, I., Fumero, J. and De Sande, F. (2012a). accull: An user-directed approach to heterogeneous programming.

Reyes, R., López, I., Fumero, J. and De Sande, F. (2012b). Directive-based programming for gpus: A comparative study.

Reyes, R., López-Rodríguez, I., Fumero, J. and De Sande, F. (2012a). accull: An openacc implementation with cuda and opencl support, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* .

.

Reyes, R., López-Rodríguez, I., Fumero, J. J. and de Sande, F. (2012b). accull: an openacc implementation with cuda and opencl support, *Euro-Par 2012 Parallel Processing*, Springer, pp. 871–882.

Reyes, R., López, I., Fumero, J. and de Sande, F. (2013). A preliminary evaluation of openacc implementations, *Journal of Supercomputing* pp. 1–13.

Sabne, A., Sakdhnagool, P. and Eigenmann, R. (2012). Effects of compiler optimizations in openmp to cuda translation, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **7312 LNCS**: 169–181. cited By (since 1996) 0.

Snyder, L. (1988). A taxonomy of synchronous parallel machines, *Technical report*, DTIC Document.

Wienke, S., Springer, P., Terboven, C. and An Mey, D. (2012a). Openacc - first experiences with real-world applications, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* .

Wienke, S., Springer, P., Terboven, C. and an Mey, D. (2012b). Openacc—first experiences with real-world applications, *Euro-Par 2012 Parallel Processing*, Springer, pp. 859–870.

Xu, R., Tian, X., Chandrasekaran, S. and Chapman, B. (2015). Multi-gpu support on single node using directive-based programming model, *Scientific Programming* **501**: 621730.

Zhang, X., Han, L., Huang, L., Li, X. and Liu, Q. (2009). A staggered-grid high-order difference method of complex frequency-shifted pml based on recursive integration for elastic wave equation, *Chinese Journal of Geophysics (Acta Geophysica Sinica)* **52**(7): 1800–1807.

Zhang, Y., Burcea, M., Cheng, V., Ho, R. and Voss, M. (2004). An adaptive openmp loop scheduler for hyperthreaded smps., *ISCA PDCS*, pp. 256–263.

.