UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ACATLÁN

Cálculo del tensor del gradiente gravimétrico a partir de un ensamble de prismas rectangulares en estaciones de trabajo multi-GPU utilizando un paradigma híbrido OPENMP/CUDA.

TESIS

QUE PARA OBTENER EL TÍTULO DE:

LIC. EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN

PRESENTA:

FELIPE ANDRÉS TORRES TORRES

ASESOR: DR. CARLOS COUDER CASTAÑEDA

Santa Cruz Acatlán, Estado de México FECHA: Abril 2019

AGRADECIMIENTOS

Antes que nada gracias al principio de causalidad por permitirme lograr este proyecto de vida.

A mi familia que siempre me a brindado su apoyo incondicional.

A la UNAM, gracias por la gran experiencia que me brindó durante mi etapa estudiantil.

Al Centro de Desarrollo Aeroespacial del Instituto Politécnico Nacional, gracias por el apoyo para poder realizar los experimentos requeridos en este trabajo.

Al Dr. Carlos Couder Castañeda, por sus enseñanzas y su gran apoyo durante este trabajo.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

La primera supercómputadora Cray-1 fue construida en 1976 por Seymor Cray, y se pensaba que podría satisfacer la demanda de cómputo requerida en ciencia e ingeniería de aquellos tiempos, sin embargo, actualmente un iPhone X tiene mayor poder de cómputo que la Cray-1.

La necesidad de cómputo que requieren los problemas actuales de computación donde se necesita exactitud y precisión (computación dura) y de inteligencia artificial, son enormes, la simulación del clima, la propagación de ondas sísmicas, la formación de galaxias, la simulación de yacimientos, la dinámica molecular, demandan cada vez más poder de cómputo a bajo costo, no obstante, con la introducción de las unidades de procesamiento gráfico (GPUs) en la computación de alto rendimiento cambio la arquitectura de los equipos y el paradigma de programación y aunque los aceleradores gráficos hacen que muchas aplicaciones puedan acelerar su ejecución a un bajo costo energético existen inconvenientes, en relación a que son relativamente caras de adquirir, relativamente más complejas de programar y sobre todo que el código no es tan fácil de migrar, es decir, está fuertemente ligado a la arquitectura para el que fue desarrollado.

A pesar de que la programación para equipos de alto rendimiento requiere una mayor complejidad debido a la introducción de los GPUs, la construcción de supercómputadoras en los últimos años integra dentro de su arquitectura GPUs, debido a que existe una necesidad inherente de incluirlas para aumentar el rendimiento a un bajo consumo energético, por ejemplo, de acuerdo a la lista de las 500 computadoras más poderosas (www.top500.org) la máquina más poderosa a Junio del 2018 es la Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, que utiliza GPUs Volta.

La introducción de los GPUs nació de la necesidad de las limitantes físicas que imponen las velocidades de frecuencia alcanzadas por materiales de los procesadores convencionales y aunque se introdujo la idea de incorporar más de un núcleo de procesamiento por procesador esta tecnología no ha sido suficiente para cubrir la demanda de cómputo, por lo que la evolución del hardware de cómputo de alto rendimiento con multiprocesadores (clusters) básicamente ha seguido dos líneas de desarrollo, la tecnología multinúcleo (multicore) como el procesador Xeon E7-8894 v4 que integra 24 cores con HiperHilado y la tecnología de los GPU (Chai et al., 2007).

Es necesario mencionar, que existen otra clase de aceleradores como los coprocesadores Xeon Phi basados en tecnologías multinúcleo, los cuales tienen la ventaja de

que el modelo de programación no cambia al conservar la misma arquitectura x86, sin embargo, no han alcanzado el poder de cómputo que ofrecen los GPUs (Teodoro et al., 2014).

La evolución de los sistemas de computo implicó un cambio en el paradigma de programación al incorporar los GPUs, el diseño de aplicaciones numéricas de propósito científico e ingenieril, tenían que rediseñarse para adaptarse a este nuevo paradigma, y aunque en los últimos 10 años NVIDIA ha tratado de mantener un estándar de programación a través de CUDA (Compute Unified Device Architecture) que incluye: los drivers, el compilador y las librerías; no obstante, cada versión nueva de CUDA exige una mayor versión del hardware, dejando obsoletos a varias arquitecturas previas, actualmente CUDA se encuentra en la versión 9 y la arquitectura de hardware más avanzanda es la Volta, representada con su tarjeta más poderosa la Tesla V100. Y aunque es relativamente caro adquirir una Tesla V100, actualmente es posible realizar programación con los GPUs que están incluidos en los equipos de escritorio y poder desarrollar aplicaciones que no demandan mucho rendimiento.

No obstante, a pesar del vertiginoso crecimiento que ha tenido la arquitectura de los GPUs, y que constantemente las interfaces de programación se actualizan, se puede considerar que se han mantenido en su mismo concepto y metodología, ya que la base de la programación del GPU desde sus origines es el kernel, el cual es un fragmento de código (codelet) que contiene una función que se ejecuta en paralelo por muchos hilos contenidos en bloques, y es la esencia de la programación en GPUs. Las aplicaciones migradas que originalmente se ejecutaban en equipos multinúcleo o cluster son demasiado vastas, se listan algunas de las aplicaciones desarrolladas que pueden ser relevantes en su campo:

Modulación directa de campos gravitacionales en MPI (Couder-Castañeda et al., 2013; Couder-Castañeda et al., 2015), reconstrucción de imágenes 3D (Zhang et al., 2014), propagación de ondas acústicas (Nakata et al., 2011), estudios de turbulencia convectiva (Calore et al., 2016), modelación de transporte radiativo (Al-Refaie et al., 2017), cómputo de estructuras Lagrangianas coherentes (Lin et al., 2017), flujos en medios porosos (Huang et al., 2015), compresión de gráficos (Kaczmarski et al., 2015), procesamiento de imágenes (Galizia et al., 2015), aceleración de consultas en bases de datos(Strohm et al., 2015), modelación en multi-física (Krol et al., 2015), resolución de las ecuaciones de transporte de Boltzmann (Priimak, 2014), aceleración de códigos de Dinámica de Fluidos Computacional (Xu et al., 2014).

La aplicación que se aborda en este trabajo es la modelación directa de la gradiometría de gravedad, la cual tiene su origen en la guerra fría y era utilizada en los submarinos de guerra para guiarse en la oscuridad, ya que los sonares podían ser detectados por el enemigo, por tal motivo, para que la navegación submarina fuera discreta, medían las variaciones minúsculas de la tracción gravitatoria causadas por

las dorsales submarinas; los gravímetros más sofisticados los portaban los submarinos estadounidenses y durante muchos años fue un secreto militar bien guardado, no obstante, ahora la tecnología se ocupa para precisar la localización de las bolsas de petróleo y de gas en las profundidades del subsuelo.

El pionero en medir la fuerza de gravedad de manera novedosa en 1890 fue el barón Roland von Eötvös, físico húngaro, que con un instrumento sencillo medía variaciones diminutas de gravedad en un lugar causadas por la presencia de un objeto cercano de masa suficiente, este primer aparato se llamó gravímetro. Actualmente los gravímetros más comunes usan resorte, pero existen de muy alta tecnología electrónicos que pueden ser utilizados en la carga útil de un nanosatélite (Hernández-Gómez et al., 2017).

Actualmente la utilización de la gravimetría es una de gran utilidad para la exploración geofísica enfocado a la minería, hidrocarburos, estudios de la corteza terrestre y fuentes geotermales. El uso de aviones y barcos para transportar gravímetros son los medios de transporte más comunes debido a que cubren áreas extensas.

Los nuevos métodos de exploración requieren calcular todo el tensor gravimétrico, el cual está conformado por las primeras derivadas del vector de gravedad en las tres direcciones ortogonales. La variación en la densidad produce las anomalías gravimétricas en el subsuelo, y la aproximación por el modelado directo consiste en discretizar el subsuelo y calcular el efecto acumulativo de cada celda discreta para calcular el efecto total. Los resultados obtenidos de manera estocástica pueden ser comparados con los datos medidos en campo y se modifica el modelo de densidades propuestos hasta que lo observado contra lo calculado coincida. No obstante, el proceso de calcular el modelo de densidades, conocido como *problema directo*, es altamente costoso computacionalmente, por lo que existen distintos trabajos que lo abordan computacionalmente para reducir el tiempo de cómputo.

La modelación directa es un requisito para muchas aplicaciones geofísicas, debido a que permite calcular las anomalías de la gravedad en modelos geológicos a escala local o regional. Trabajos relacionados pueden verse en (Chen and Zhang, 2018), donde se lleva a cabo el cálculo del modelo directo en un solo GPU con unión de celdas; el cálculo directo en un solo GPU puede consultarse en (Moorkamp et al., 2010a), el cálculo directo en multi-GPU utilizando OpenACC (Couder-Castaneda et al., 2013) y utilizando aceleradores Xeon Phi (Arroyo et al., 2015). La diferencia esencial de los trabajos encontrados que existen en la literatura y el elaborado en esta tesis, es que se propone un diseño basado en prismas convencionales y paralelizado en múltiples GPUs bajo herramientas 100 % gratuitas, sin el uso de directivas como Openacc, lo cual facilita su migración relativamente fácil entre compiladores gratuitos.

JUSTIFICACIÓN

El tiempo del cálculo del tensor del gradiente gravimétrico tiene relevancia en las áreas de geofísica, donde se utilizan en la exploración de los hidrocarburos, la distribución de cuerpos salinos o someros e incluso modelos sísmicos, este tipo de problemas en su mayoría tienen que procesar una gran cantidad de datos y realizar los cálculos en el menor tiempo posible.

Para la aplicación desarrollada en este trabajo existe una versión previa, implementada utilizando la librería de paso de mensajes (MPI) como control de distribución, la ventaja de usar MPI, es que permite utilizar a la unidad de procesamiento de gráficos (GPU) que están distribuidos en el cluster de computadoras, no obstante, los inconvenientes es que los GPUs que se encuentran integrados en la misma plataforma, no sacan provecho de la velocidad que les puede proporcionar la placa base en la que están integrados. Por tal motivo, se enfocó en desarrollar una aplicación para la modelación directa de campos gravitacionales para plataformas multi-GPU integradas en la misma placa base y de uno a tres GPUs, con la finalidad de obtener el máximo rendimiento.

El tensor del gradiente gravimétrico se calculará a partir de una integración de las API (Application programming interface) OpenMP y CUDA y se construyen dos estructuras, una bidimensional y una tridimensional para realizar el ensamble de prismas del gradiente gravimétrico para con ello mostrar que es posible reducir los tiempos de cómputo en comparación con la ejecución en un CPU convencional y con esto mejorar el rendimiento de la aplicación.

OBJETIVO

Diseñar e implementar un código híbrido OpenMP/CUDA en lenguaje C para calcular el tensor del gradiente gravimétrico a partir de un ensamble de prismas rectangulares.

Los objetivos particulares son:

- Crear una estructura de datos bidimensional en CUDA que permita el manejo de arreglos bidimensionales para el manejo de la malla de observaciones.
- Desarrollar una estructura de datos tridimensionales en CUDA que permita el manejo de arreglos tridimensionales para el manejo del ensamble de prismas.
- Migrar las funciones que calculan el tensor a kernels CUDA.
- Validar con ejemplos conocidos los cálculos.

- Diseñar el código utilizando OpenMP como controlador y CUDA como paralelizador.
- Llevar a cabo pruebas de rendimiento desde un GPU hasta 3 GPUs.

HIPÓTESIS

Es posible reducir el tiempo de cómputo del cálculo directo del tensor del gradiente gravimétrico a partir de un ensamble de prismas rectangulares utilizando GPUs integrados en la misma estación de trabajo.

1

ARQUITECTURA DE LOS CPU Y GPU

Se introducen las arquitecturas de memoria compartida, los sistemas multi-núcleo y los GPU de NVIDIA.

1.1 ANTECEDENTES

Desde tiempos antiguos el hombre ha necesitado hacer cálculos. Ésto lo ha llevado a construir artefactos que lo ayuden a realizarlos de una manera más fácil y rápida. Algunas de las herramientas que se han construido a lo largo del tiempo son: el ábaco, el cuadrante,tablas de Neper, las reglas de cálculo, entre otros.

A partir del siglo XVII se realizaron progresos mecánicos en los sistemas de cálculo. Aparecen las calculadoras mecánicas, como la máquina de Pascal y la máquina Leibnitz.

En 1820, Charles Babbage, construye la máquina analítica, algunas de sus características son:

- Trabajaba con una aritmética de 50 dígitos decimales.
- Consiguió mejorar los tiempos de los cálculos aritméticos a un segundo para sumar y restar y un minuto para multiplicar y dividir.
- Trabajó con tarjetas perforadas para indicar las operaciones a realizar y las variables.

En 1943, Mauchly y Eckert construyeron ENIAC (Electronic Numeric Integrator and Calculator). Fue considerada la primer computadora digital electrónica. Esta computadora estaba orientada a resolver problemas de carácter científico.

En los años siguientes fueron apareciendo computadoras para uso personal como la Comodore Pet, Apple II, TR-80, todas estas maquinas contaban con un solo procesador.

La constante demanda de las nuevas aplicaciones hizo que la industria de los monoprocesadores se encontrara en una situación límite respecto al cumplimiento de la Ley de Moore que afirmó que el número de transistores en un microprocesador se duplicaría cada año, aunque posteriormente en 1975 esta ley se modificó y predijo que el ritmo bajaría aproximadamente cada 18 meses. Como los monoprocesadores

alcanzaron su rendimiento máximo, se comenzaron a buscar otras alternativas. Una de ellas fueron los multiprocesadores, computadoras de propósito general con dos o mas núcleos.

Al mismo tiempo que los multiprocesadores eran aceptados por la sociedad en general y ante el auge de la industria de los videojuegos, grandes y relevantes avances tecnológicos fueron hechos en las Unidades de procesamiento de gráficos (GPU), con el objetivo de liberar a la unidad central de procesamiento (CPU) del proceso de renderizado, propio de las aplicaciones gráficas. La gran demanda de los gráficos de alta calidad motivó el incremento de la potencia de cálculo transformando a las GPU en potentes coprocesadores paralelos. Si bien su origen fue brindar asistencia en aplicaciones gráficas, su uso como co-procesador paralelo de la CPU para resolver aplicaciones de propósito general constituye uno de los tópicos más actuales en la computación de alto desempeño.

A partir del 2005, se comenzó a utilizar la gran potencia de cálculo y el alto número de procesadores de las GPU como arquitectura masivamente paralela para resolver tareas no vinculadas con actividades gráficas, es decir utilizarlas en aplicaciones de propósito general. En este ámbito surgieron varias técnicas, lenguajes y herramientas para la programación de GPU como co-procesador genérico a la CPU. La evolución de estas fue tan rápida como su popularización. Una de las herramientas más difundidas es CUDA que es una arquitectura de cálculo paralelo desarrollada por la compañía NVIDIA.

1.2 ARQUITECTURA CPU

La arquitectura de computadoras se refiere a los atributos de un sistema que son visibles a un programador, en otras palabras, son aquellos atributos que tienen un impacto directo en la ejecución lógica de un programa. La organización de las computadoras se refiere a las unidades funcionales y sus interconexiones, que dan lugar a especificaciones arquitectónicas, como ejemplo tenemos el número de bits usados para representar varios tipos de datos (números, caracteres, booleanos), mecanismos de entrada y salida (E/S) y técnicas para direccionamiento de memoria.

Los componentes de una computadora y sus funciones son:

- Procesador: Se encarga de gestionar y controlar las operaciones.
- Memoria: Almacena información (los programas y los datos necesarios para ejecutarlos).
- Sistema de E/S: Permite la comunicación entre el usuario y la computadora, permitiendo introducir información y desplegar resultados.

- Sistema de interconexión: Proporciona los mecanismos necesarios para interconectar todos los componentes.

En 1945 Von Neumann propuso el modelo de programa almacenado de la computación, donde propone que un programa es una secuencia de instrucciones que son ejecutadas secuencialmente (Von Neumann and Mauchly, 1945).

La mayoría de las computadoras se han construido siguiendo la arquitectura de Von Neumann que cuenta con la estructura:

- Entrada: Unidad que transmite instrucciones y datos del exterior a la memoria (pasando por la ALU).
- Memoria: Unidad que almacena instrucciones y datos, así como los resultados parciales y finales de los programas.
- ALU: Unidad que realiza las operaciones aritmético-lógicas (suma, multiplicación, resta y operaciones lógicas).
- Unidad de control: Interpreta las instrucciones y coordina el resto del sistema.
- Salida: Transmite los resultados al exterior.

El funcionamiento de la arquitectura parte de las instrucciones máquina que se almacenan en memoria, accediendo mediante direcciones y ejecutando los siguientes pasos:

- Búsqueda de la instrucción.
- Decodificación.
- Cálculo de la dirección de los operandos.
- Búsqueda de los operandos.
- Ejecución (realiza la operación y almacena el resultado).

Cada instrucción máquina debe especificar:

- La operación a realizar en el código de operación.
- Información para calcular las direcciones de los operandos y dónde se guarda el resultado.
- Información de la dirección de la próxima instrucción a ejecutar.

1.3 RENDIMIENTO CPU

La mayoría de las computadoras se construyen utilizando un reloj que funciona a una frecuencia constante (ciclos de reloj). El tiempo, es la medida del rendimiento de

una computadora. La ejecución de un programa se mide en segundos, el rendimiento se mide como una frecuencia de eventos por segundo, ya que un menor tiempo de ejecución significa mayor rendimiento.

El tiempo de un ciclo de reloj se puede referenciar por su duración, por ejemplo: 10 nano segundos, 100 MHZ. Por lo tanto, podemos expresar el tiempo del CPU como:

Tiempo de CPU = Ciclos de reloj de CPU para un programa $\times$ la duración del ciclo de reloj.

Además del número de ciclos de reloj para ejecutar un programa, también podemos contar el número de instrucciones ejecutadas. Si conocemos el número de ciclos de reloj y el número de instrucciones ejecutadas podemos calcular el número medio de ciclos de reloj por instrucción (CPI):

CPI = Ciclos de reloj de CPU para un programa / Número de instrucciones ejecutadas

Sin embargo hay que tener en cuenta que la medida real del rendimiento del CPU es el tiempo.

1.4 LEY DE AMDAHL

El término cuello de botella se utiliza para referirse al subsistema o subsistemas que degradan el rendimiento del equipo en general. Dado que todos los componentes de una computadora están interconectados, un cambio en un subsistema tiene un impacto inmediato en el rendimiento del sistema en general.

Ahora bien, estos conceptos pueden ser llevados al rendimiento de un programa. Si se requiere mejorar el rendimiento de un programa, lo mejor es enfocarse en mejorar rutinas donde se tienen identificados problemas de rendimiento.

Si las rutinas mejoran se pueden bajar los tiempos de ejecución. Como métrica se emplea la aceleración o ganancia en tiempos, conocida como speed-up, que se define como el cociente del tiempo empleado por el sistema a evaluar y una referencia:

$$\alpha = \frac{\text{Tiempo consumido}}{\text{Tiempo de referencia}}$$

Para cuantificar el rendimiento global de un sistema a partir del rendimiento de sus partes se utiliza la expresión de la ley de Amdhal, que establece que la mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.

De forma analítica, esta ley se expresa de la siguiente manera:

$$\alpha = \frac{1}{(1-p) + \frac{p}{\alpha_m}}$$

donde:

α: Representa el cociente entre el tiempo de ejecución completo del programa antes de mejorar alguno de sus componentes y el tiempo de ejecución después de ser mejorado dicho componente.

α_m: Representa el factor de mejora que se ha introducido en el subsistema alterado.

p: Es la fracción de tiempo que, del sistema completo original utiliza el subsistema que se ha alterado (es decir, cuánto se usa el componente que se ha mejorado en el sistema original).

1.5 ORGANIZACIÓN DE UN SISTEMA MULTIPROCESADOR

En la organización de un sistema multiprocesador hay dos o más procesadores. Cada procesador es autónomo, incluyendo una unidad de control, una ALU, registros y caché. Cada procesador tiene acceso a una memoria principal compartida y a los dispositivos de E/S a través de alguna forma de mecanismo de interconexión. Los procesadores pueden comunicarse entre sí a través de la memoria. También es posible que los procesadores intercambien señales directamente. La memoria a menudo se organiza de forma que sean posibles los accesos simultáneos a bloques de memoria separados.

La organización más común en las PC, estaciones de trabajo y servidores es el bus de tiempo compartido. El bus de tiempo compartido es el mecanismo más simple para construir un sistema multiprocesador. La estructura y las interfaces son básicamente las mismas que las de un sistema de único procesador, que utilice un bus para la interconexión. El bus consta de dos líneas de control, dirección y datos. Para facilitar las transferencias se proporcionan los siguientes elementos:

- Direccionamiento: Debe ser posible distinguir los módulos del bus para determinar la fuente y el destino de los datos.

- Arbitraje: Se proporciona un mecanismo para gestionar las peticiones que compiten por el control del bus, utilizando algún tipo de esquema de prioridad.
- Tiempo compartido: Cuando un módulo está controlando el bus, los otros módulos no tienen acceso al mismo y deben, si es necesario, suspender su operación hasta que se disponga del bus.

En este caso, hay varias CPU además de varios procesadores que intentan tener acceso a uno o más módulos de memoria a través del bus.

Algunas de las ventajas de la organización del bus son:

- Simplicidad: La interfaz lógica y física de cada procesador para el direccionamiento, el arbitraje y para compartir el tiempo del bus, es el mismo que el de un sistema con un solo procesador.
- Flexibilidad: Se pueden conectar más procesadores al bus.
- Fiabilidad: El fallo de cualquiera de los dispositivos conectados no provocara el fallo de todo el sistema.

Sin embargo, la desventaja del bus son las prestaciones. Es decir, todas las peticiones a memoria pasan por el bus. En consecuencia, la velocidad del sistema está limitada por el tiempo de ciclo. Para mejorar las prestaciones, se puede equipar a cada procesador de una memoria caché, con lo cual se reduciría el número de accesos. Normalmente las PC's y las estaciones de trabajo tienen dos niveles de caché, un caché L1 interna (en el mismo chip que el procesador) y una cache L2 externa o interna.

El uso de cachés introduce algunas consideraciones nuevas, puesto que cada caché local contiene una imagen de una parte de la memoria, si se altera una palabra en un caché, podría invalidar una palabra en otro caché. Para evitarlo, se debe avisar a los otros procesadores de que se ha producido una actualización de memoria.

1.6 CONCURRENCIA

La concurrencia en el software es una forma de gestionar los recursos compartidos usados al mismo tiempo. Un término utilizado en la concurrencia es el termino de hilo (thread), un hilo tiene la propiedad de poder compartir recursos entre sí, por ejemplo, el rango de direcciones de memoria asignadas. Podemos definir a un hilo como una secuencia discreta de instrucciones relacionadas que se ejecuta independientemente de otras secuencias de instrucciones. Cada programa tiene al menos un hilo principal, que inicializa el programa y comienza a ejecutar las instrucciones, el hilo principal puede crear otros hilos que realizan varias tareas, o simplemente puede hacer todo el trabajo en sí sin crear más hilos. (Jason Roberts, 2006).

El modelo computacional de hilo cuenta con tres niveles:

- Hilos de nivel de usuario: Hilos creados y manipulados en el software de la aplicación.
- Hilos a nivel de kernel: La forma en que el sistema operativo implementa la mayoría de los hilos.
- Hilos de hardware: Cómo aparecen los hilos en los recursos de ejecución en el hardware.

Algunas ventajas que podemos observar de la concurrencia en el software son las siguientes:

- Permite mayor eficiencia en el uso de los recursos del sistema. El uso eficiente de la utilización de los recursos es la clave para maximizar el rendimiento de los sistemas computacionales.
- Proporciona una abstracción para la implementación de algoritmos de software o aplicaciones que están en paralelo.

Cuando múltiples hilos de ejecución están corriendo en paralelo, significa que los hilos activos están ejecutándose simultáneamente en diferentes recursos del hardware, es decir, múltiples hilos pueden hacer progreso simultáneo. Cuando múltiples hilos de ejecución del software están ejecutándose concurrentemente, las ejecuciones de los hilos son intercaladas dentro de un sólo recurso del hardware.

En los sistemas multitarea, un proceso se puede encontrar en tres estados distintos (Asenjo Plaza Rafael, 2001):

- En ejecución: Está siendo atendido por la CPU, o usando la CPU.
- Listo: El proceso está libre para ser ejecutando.
- Bloqueado: Está a la espera de que ocurra algún evento, normalmente alguna transacción de E/S.

En sistemas monoprocesador sólo puede haber un proceso en ejecución, pero varios listos y bloqueados. El sistema operativo tiene una lista con los procesos listos (ordenados por prioridad) y otra con los procesos bloqueados (en este caso la lista está desordenada).

Las transiciones entre estos estados se producen mediante llamadas del sistema operativo a una serie de funciones, con el nombre del proceso como parámetro, excepto en el caso del paso de listo a bloqueado, que es provocada por el propio proceso cuando necesita realizar una operación de E/S.

1.7 SISTEMAS PARALELOS

Por definición, una arquitectura paralela es aquella que cuenta con varias unidades de procesamiento y permite el procesamiento paralelo. La disposición de las conexiones entre los procesadores y la memoria determina el origen de distintas arquitecturas, las cuales serán más adecuadas para resolver ciertos problemas.

La construcción de una aplicación en paralelo no siempre deriva naturalmente de la aplicación secuencial. En toda aplicación paralela se deben considerar las comunicaciones existentes entre los distintos procesos, las cuales pueden conducir a un mal desempeño del programa.

En sistemas paralelos se pueden distinguir dos conceptos: el paralelismo de las instrucciones y el paralelismo de la máquina

- Paralelismo en las instrucciones: Se produce cuando las instrucciones de una secuencia son independientes y por tanto pueden ejecutarse en paralelo, también depende de la frecuencia de dependencias de datos que haya en el código, estos factores dependen a su vez de la arquitectura del conjunto de instrucciones y de la aplicación. El paralelismo en las instrucciones depende también de lo que se llama latencia de una operación, es decir, el tiempo que transcurre hasta que el resultado de una instrucción está disponible para ser usado como operando de una instrucción posterior, la latencia determina cuánto retraso causará una dependencia de datos.
- El paralelismo de la máquina: Es una medida de la capacidad del procesador para sacar partido al paralelismo de las instrucciones. El paralelismo de la máquina depende del número de instrucciones que pueden captarse y ejecutarse al mismo tiempo, así como de la velocidad y sofisticación de los mecanismos que usa el procesador para localizar instrucciones independientes.

El paralelismo en las instrucciones, como el paralelismo de la máquina, son factores importantes para mejorar el uso de los recursos. Un programa puede no tener el suficiente nivel de paralelismo en las instrucciones como para sacar el máximo partido al paralelismo de la máquina.

El empleo de una arquitectura con instrucciones de longitud fija, como en una arquitectura computacional con un conjunto de instrucciones reducidas (RISC), aumenta el paralelismo en las instrucciones ya que posibilita la segmentación y el paralelismo en la ejecución de instrucciones, así como reducir los accesos a la memoria. Por otra parte, un escaso paralelismo de la máquina limitará las prestaciones sin que importe la naturaleza del programa.

1.8 PLATAFORMAS DE CÓMPUTO PARALELAS

La construcción del software paralelo se basa en la idea de dividir un problema grande en subproblemas, cada uno de los cuales se resuelve en forma concurrente al resto.

Implementar algoritmos paralelos que se ejecuten sobre máquinas paralelas no es una tarea sencilla, ya que no existe una solución paralela única. Para determinar si un programa puede paralelizarse se debe analizar la naturaleza del problema y la relación entre los datos, para el análisis se pueden realizar los siguientes puntos:

- Dividir el trabajo en tareas.
- Asignar tareas a los distintos procesadores.
- Ver la relación de los procesadores.
- Verificar como se organizan las comunicaciones.

Cada uno de estos aspectos, que, si bien aparentan ser independientes no lo son, las decisiones adoptadas en uno influyen directamente en los otros. Estos problemas se pueden resolver si se utiliza una metodología para los desarrollos.

Para lograr la ejecución de software paralelo, el hardware debe proporcionar una plataforma que soporte la ejecución simultánea de múltiples hilos. La arquitectura de computadoras puede ser clasificada por dos diferentes dimensiones. La primera es el número de instrucciones en el flujo que deberán ser procesadas en un solo punto en el tiempo. La segunda es el número de datos que pueden procesarse en un solo punto en el tiempo. A este sistema de clasificación se le conoce como la taxonomía de Flynn (Flynn, 1972), y es representado en la Figura 1.1.

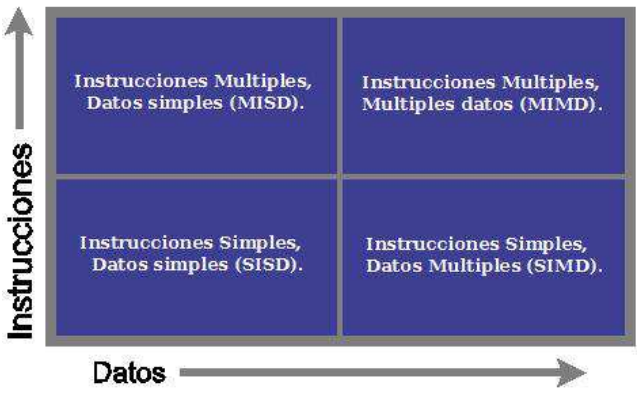


Figura 1.1: Taxonomía de Flynn

La taxonomía de Flynn cuenta con cuatro categorías:

- **Una sola instrucción un único dato (SISD):** No se ofrece paralelismo en el hardware. Las instrucciones se ejecutan de manera secuencial. Solo un flujo de datos es procesado por el CPU durante un ciclo de reloj dado.
- **Múltiples instrucciones un único dato (MISD):** Se procesa un único flujo de datos a través de varias secuencias de instrucciones de forma simultánea. En la mayoría de los casos, múltiples flujos de instrucciones necesitan múltiples flujos de datos para ser útil, por lo que en general esta clase de computadora paralela se utiliza más como un modelo teórico que de práctica.
- **Una sola instrucción múltiples datos (SIMD):** Un flujo de instrucciones simples tiene la capacidad de procesar múltiples flujos de datos simultáneamente. Estas máquinas son útiles en aplicaciones de tratamiento de señales digitales, procesamiento de imágenes y las aplicaciones multimedia, como audio y video.
- **Múltiples instrucciones múltiples datos (MIMD):** Se tiene la capacidad de ejecutar múltiples flujos de instrucciones, mientras se trabaja en un flujo de datos separado e independiente. Esta es la plataforma de computación en paralelo más común en estos días. Nuevas plataformas como los *clusters* (conjunto de computadoras que se comportan como una sola) caen en esta categoría.

Dado que las máquinas actuales encajan en las categorías SIMD o MIMD, se puede explotar a nivel de datos y nivel de tareas el paralelismo en el software.

La tecnología SMT (Simultaneous Multithreading) permite que un procesador aparezca como múltiples procesadores lógicos. Desde el enfoque de la microarquitectura, las instrucciones de los procesadores lógicos son persistentes y se ejecutan simultáneamente en recursos de ejecución compartida. En otras palabras, se pueden programar múltiples hilos, pero como los recursos de ejecución se comparten la microarquitectura debe determinar cómo y cuándo intercalar la ejecución de los hilos. El siguiente nivel después del procesamiento Multi-Hilado es el procesamiento Multi-Núcleo, estos núcleos cuentan con sus propios recursos de ejecución y de arquitectura, dependiendo del diseño estos procesadores pueden compartir un gran cache en el chip además estos núcleos individuales pueden combinarse con SMT aumentando el número de procesadores lógicos por el doble de núcleos de ejecución.

1.9 EVOLUCIÓN DE LA GPU

La historia de las GPU se inicia en la década de los 60. Las primeras GPU tuvieron capacidades muy reducidas, sin embargo, su crecimiento no se detuvo. Así como la velocidad del procesador avanzó a través de dos líneas: incrementando la velocidad del reloj y/o incrementando el número de núcleos, la evolución de las GPU tuvo su

origen a finales de los 80 cuando se inicia la gran demanda de mejores interfaces gráficas por parte de los sistemas operativos. Esto implicó que a principios de los 90 se comenzaran a vender aceleradores gráficos 2D para computadoras personales.

Al mismo tiempo, en la comunidad profesional y mediante la empresa Silicon Graphics, se introdujó al mercado el uso de gráficos 3D en una gran variedad de ámbitos. Además, se desarrolló la librería OpenGL para ser usada como método independiente de la plataforma y poder escribir aplicaciones gráficas 3D. La demanda de aplicaciones gráficas 3D tuvo un gran crecimiento, el cual fue alentado primero, por el desarrollo de videojuegos en primera persona, FPS (Harrigan., 2004) como Doom, Duke Nukem 3D y Quake, los cuales demandaron constantemente mayores capacidades para lograr mejor realismo.

En 1999 NVIDIA lanza al mercado la tarjeta gráfica Geforce 256, la cual permitía realizar transformaciones e iluminación a través del hardware, además de brindar mejores condiciones de visualización. La siguiente generación de NVIDIA constituyó un gran paso en la tecnología de las GPU, fue considerada la primer GPU con implementación nativa de la primera versión de la interfaz de programación de aplicaciones (API) DirectX8. Esto dio paso a que por primera vez los desarrolladores tuviesen el control en la GPU.

Desde 1999 hasta 2002, NVIDIA dominó el mercado de las tarjetas gráficas con las GeForce. En ese período, las mejoras se orientaron hacia el campo de los algoritmos 3D y la velocidad de los procesadores gráficos. Las memorias también necesitaban mejorar la velocidad, por lo que se incorporaron las memorias del tipo DDR (Double Data Rate) a las tarjetas gráficas (Bruce Jacob, 2007). Las capacidades de memoria de vídeo pasaron de los 32 MB de las Geforce 2 a los 64 y 128 MB de la GeForce 4. A partir del 2006, NVIDIA y ATI tomaron el liderazgo del mercado con sus series GeForce y Radeon, respectivamente.

1.10 PIPELINE GRÁFICO

La GPU desde sus inicios fue un procesador con muchos recursos computacionales. Actualmente ha adquirido notoriedad por su uso en aplicaciones de propósito general, pasó de ser un procesador con funciones especiales a ser considerado la arquitectura base de aplicaciones paralelas y se han convertido en una parte integral de los sistemas actuales de computación.

En los últimos años, su evolución implicó un cambio, dejó de ser un procesador gráfico potente para convertirse en un co-procesador apto para el desarrollo de aplicaciones paralelas de propósito general con demanda de anchos de banda, de procesamiento y de memoria superiores a los ofrecidos por la CPU.

Para poder entender a la GPU como una unidad de procesamiento de propósito general (GPGPU) es necesario comprender su funcionamiento desde el punto de vista del hardware, es decir, los gráficos. Esto permitirá realizar una analogía de cada uno de los mecanismos propios con los que se aplican en GPGPU.

La GPU trabaja cuando la aplicación envía a la GPU una secuencia de vértices, agrupados en lo que se denominan primitivas geométricas (polígonos, líneas y puntos) y son tratadas secuencialmente a través de cuatro etapas:

- Transformación de vértices: Es la primera etapa del pipeline de procesamiento gráfico. En ella se lleva a cabo una secuencia de operaciones matemáticas sobre cada uno de los vértices suministrados por la aplicación.
- Ensamblado de primitivas y rasterización: Los vértices generados y transformados en la etapa anterior pasan a esta segunda etapa, donde son agrupados en primitivas geométricas basándose en la información recibida junto con la secuencia inicial de vértices. Como resultado, se obtiene una secuencia de triángulos, líneas o puntos. La rasterización es el proceso por el cual se determina el conjunto de píxeles "cubiertos" por una primitiva determinada. Los resultados de la rasterización son conjuntos de localizaciones de píxeles y conjuntos de fragmentos. Un fragmento tiene asociada una localización de píxel e información relativa a su color y uno o más conjuntos de coordenadas de textura. Se puede pensar en un fragmento como en un "píxel en potencia": si el fragmento supera con éxito el resto de etapas del pipeline, se actualizará la información de píxel como resultado.
- Interpolación, texturas y colores: Una vez hecha la rasterización, el/los fragmentos obtenidos son sometidos a operaciones de interpolación, operaciones matemáticas y de textura y determinación del color final de cada fragmento. Además de establecer el color final del fragmento, en esta etapa es posible descartar un fragmento determinado para impedir que su valor sea actualizado en memoria; por tanto, esta etapa emite uno o ningún fragmento actualizado para cada fragmento de entrada.
- Operaciones Raster: En esta última etapa del pipeline se realizan las operaciones llamadas raster, analizan cada fragmento, sometiéndolo a un conjunto de pruebas relacionadas con aspectos gráficos del mismo. Estas pruebas determinan los valores que tomará el píxel generado en memoria a partir del fragmento original. Si cualquiera de estas pruebas falla, es en esta etapa cuando se descarta el píxel correspondiente, y por tanto no se realiza la escritura en memoria del mismo. En caso contrario, y como último paso, se realiza la escritura en memoria, denominada framebuffer (Godse., 2009), como resultado final del proceso.

El paradigma paralelo estuvo presente desde los inicios de la GPU. En el pipeline gráfico la entrada de un estado es la salida del anterior, en este caso el paralelismo es a nivel de tareas, varios datos pueden estar al mismo tiempo en distintos estados del pipeline. Los recursos son divididos entre las distintas etapas del pipeline, por lo que varias unidades de cómputo trabajan en paralelo en los diferentes estados del pipeline. En otras palabras, los recursos en la GPU se dividen en el espacio. Esto implica un nivel de paralelismo interno en la etapa y otro entre las etapas del pipeline.

1.11 ARQUITECTURA GPU

Algunas de las características básicas de la GPU son las siguientes:

- Siguen el modelo SIMD. Todos los núcleos ejecutan a la vez una misma instrucción, por lo tanto, solo se necesita decodificar la instrucción una única vez para todos los núcleos.
- La velocidad de ejecución se basa en la explotación de la localidad de los datos, tanto la localidad temporal (cuando accedemos a un dato, es probable que se vuelva a utilizar el mismo dato en un futuro cercano) como la localidad espacial (cuando accedemos a una fecha, es muy probable que se utilicen datos adyacentes a los ya utilizados en un futuro cercano y por eso, se utilizan memorias caché que guardan varios datos en una línea del tamaño del bus).
- La memoria de un GPU se organiza en varios tipos de memoria (local, global, constante y textura), que tienen diferentes tamaños, tiempos de acceso y modos de acceso (por ejemplo, solo lectura o lectura/escritura).
- El ancho de banda de la memoria es mayor.

Una GPU está altamente segmentada, lo que indica que posee gran cantidad de unidades funcionales. Estas unidades funcionales se pueden dividir principalmente en dos: aquéllas que procesan vértices, y aquéllas que procesan píxeles. Por tanto, se establecen el vértice y el píxel como las principales unidades que maneja la GPU.

Inicialmente, a la GPU le llega la información de la CPU en forma de vértices. El primer tratamiento que reciben estos vértices se realiza en el vertex shader. Aquí se realizan transformaciones como la rotación o el movimiento de las figuras. Tras ésto, se define la parte de estos vértices que se va a ver, y los vértices se transforman en píxeles mediante el proceso de rasterización. Estas etapas no poseen una carga relevante para la GPU.

Donde se encuentra el principal *cuello de botella* de la GPU es en el siguiente paso: el píxel shader. Aquí se realizan las transformaciones referentes a los píxeles, tales como la aplicación de texturas.

El modelo píxel shader se convirtió en un referente, el cual permitió el desarrollo de aplicaciones sobre GPU, incrementando la complejidad de las operaciones de vértice y fragmentos. La evolución de la arquitectura de las GPU se centró principalmente en las etapas del pipeline gráfico.

1.12 DIFERENCIAS ENTRE EL CPU Y EL GPU

Una diferencia importante entre un CPU y un GPU es la comparación entre la forma en que manejan el procesamiento de tareas. Un CPU está compuesto de varios núcleos optimizados para el procesamiento en serie, mientras que un GPU está formado de miles de núcleos más pequeños y eficientes los cuales están diseñados para realizar múltiples tareas simultáneamente, ver Figura 1.2.

Las GPU poseen miles de núcleos que procesan las cargas de trabajo de forma paralela y muy eficiente.

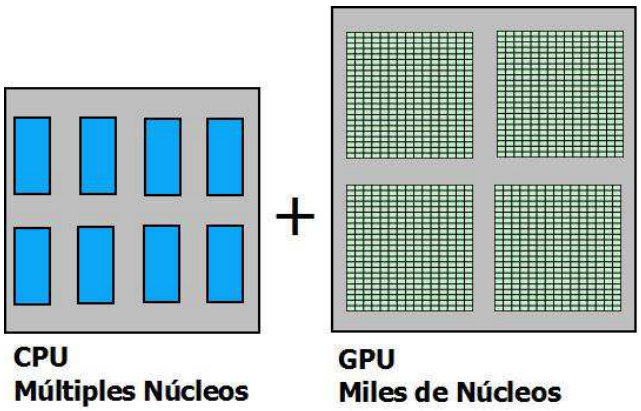


Figura 1.2: Diferencias entre el número de núcleos del CPU y el GPU.

Una de las mayores diferencias con la CPU recae en su arquitectura. A diferencia del CPU, que tiene una arquitectura de Von Neumann, la GPU se basa en el Modelo Circulante. Este modelo facilita el procesamiento en paralelo, y la gran segmentación que posee la GPU para sus tareas.

En síntesis, el modelo convencional se concentra en el control de las instrucciones, mientras que el circulante lo hace en el ancho de banda de los datos.

Debido a la organización de las GPU respecto a la CPU, hay que tener en cuenta que una GPU no puede acceder directamente a la memoria principal y que una CPU no puede acceder directamente a la memoria de una GPU. Por lo tanto, habrá que copiar los datos entre CPU y GPU de manera explícita es decir de la CPU a la GPU y viceversa.

La Figura 1.3 muestra la relación que se tiene entre el GPU y el CPU es decir cuando el código de una aplicación es lanzado desde el CPU debe tener indicadas secciones específicas que indicaran que esa parte del código debe ser ejecutado en el GPU.

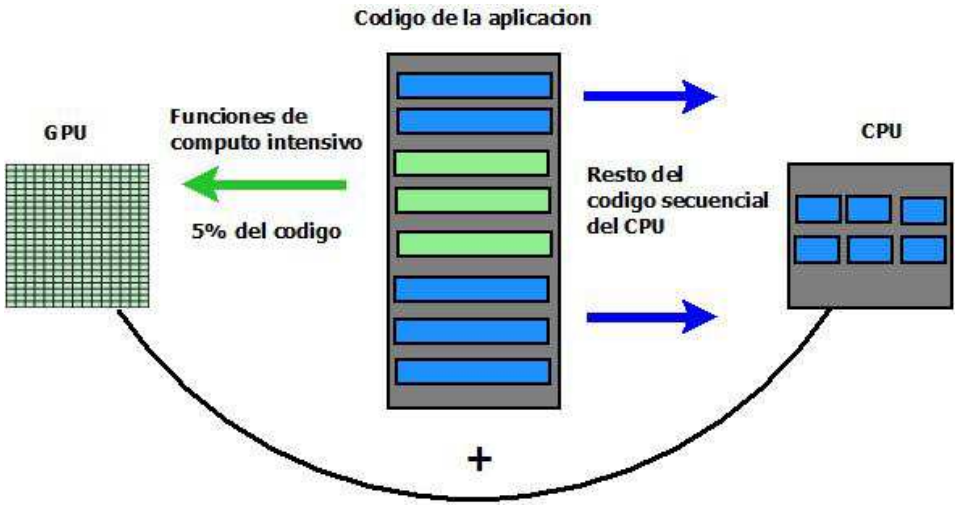


Figura 1.3: Flujo de un código de aplicación ejecutado en paralelo

2

PARADIGMA DE LA PROGRAMACIÓN EN OPENMP Y CUDA

En este capítulo se aborda las metodologías de programación utilizadas, OpenMP y CUDA, resaltando sus ventajas y considerando sus desventajas.

2.1 LA ARQUITECTURA CUDA

Las unidades de procesamiento gráfico (GPU) implementan en hardware el pipeline gráfico, el cual, al ser un algoritmo paralelo con cómputo intensivo, determinó la evolución de las GPU al enfocar los avances en satisfacer la demanda de mejor rendimiento en la ejecución del pipeline. Ésto unido a su bajo costo hace que cualquier computadora personal cuente con una GPU. Sin embargo, en un principio su programación no era tan simple, implicaba conocer detalles de la arquitectura del pipeline gráfico y utilizar interfaces de programación de aplicaciones (API) como OpenGL (Wright., 2007) o DirectX (Root., 1999).

En el año 2006 NVIDIA presentó la tecnología *Compute Unified Device Architecture* (CUDA) para su última generación de tarjetas gráficas, la serie G80. CUDA propone una filosofía integradora, con un lenguaje de programación como es C y la arquitectura paralela de una GPU, desvinculándose además del pipeline gráfico.

La tecnología CUDA permite considerar a la GPU como una arquitectura paralela para la resolución de problemas de propósito general. El desarrollo de dichas aplicaciones paralelas es posible principalmente, por dos razones:

- Las tarjetas gráficas NVIDIA, son un componente común en la mayoría de las computadoras personales actuales.
- Es de fácil aprendizaje, más para aquellos programadores con conocimientos de los lenguajes tipo C o C++.

La arquitectura de CUDA es realmente particular en muchos aspectos. A diferencia de otros GPU, CUDA no sólo divide los recursos de computación en vertex y pixel shaders, además añade un pipeline para shaders unificado que permite a cada unidad aritmético lógica (ALU) en el chip ser “transformada” por una aplicación para efectuar operaciones de cómputo de propósito general.

CUDA presenta a la arquitectura de la GPU como un conjunto de multiprocesadores MIMD (Múltiples Instrucciones-Múltiples Datos). Cada multiprocesador posee

un conjunto de procesadores SIMD (Instrucción Simple-Múltiples Datos). Respecto a la memoria, existen numerosos modelos conviviendo en esta arquitectura: cada procesador SIMD posee una serie de registros a modo de memoria local (sólo accesible por el procesador) a su vez cada multiprocesador posee una memoria compartida (accesible por todos los procesadores SIMD del multiprocesador) y finalmente la memoria global, la cual es accesible por todos los multiprocesadores y por ende, por todos y cada uno de los procesadores SIMD.

2.2 PLATAFORMA DE CÁLCULO PARALELO CUDA

No todos los problemas pueden ser resueltos en la GPU, los más adecuados son aquellos que pueden resolverse mediante la aplicación del paradigma paralelo de datos, es decir aplican la misma sentencia o secuencia de código a todos los datos de entrada. Se puede decir que una solución de un problema en GPU será más ventajosa respecto a la solución en la CPU si la aplicación tiene las siguientes propiedades:

- El algoritmo tiene un orden de ejecución cuadrático o superior: el tiempo necesario para realizar la transferencia de datos entre la CPU y la GPU tiene un gran costo, el cual no suele verse compensado por el bajo costo computacional de un método lineal.
- Es mayor la carga de cálculo computacional en cada hilo: de nuevo para compensar el tiempo de transferencia de información es conveniente que cada hilo posea una carga computacional considerable.
- Es menor la dependencia entre los datos para realizar los cálculos, ésto es posible si cada SM sólo necesita de los datos de su memoria local o compartida y no necesita acceder a memoria global, la cual tiene un acceso más lento.
- Es menor la transferencia de información entre CPU y GPU. La situación óptima es cuando la transferencia sólo se realiza una vez, al comienzo y al final del proceso. Esto significa una transferencia de los datos de entrada, desde la CPU a la GPU y una al final, desde la GPU a la CPU, para obtener los resultados. Es bueno no tener transferencias intermedias, ya sea de resultados parciales o datos de entradas intermedios.
- No existen secciones críticas, es decir, varios procesos no necesitan escribir en las mismas posiciones de memoria, las lecturas de memoria global y compartida puede ser simultánea, pero las escrituras en la misma posición de memoria plantean un acceso a un recurso compartido, lo cual implica contar con mecanismo de acceso seguro. Este proceso hace más lenta la solución del proceso global.

Además, es necesario que las estructuras de datos en la aplicación que se ejecuta en la CPU se adapten o puedan transformarse a estructuras más simples del tipo matriz o vector a fin de poder ser compatibles con las estructuras que maneja la GPU.

El modelo de programación CUDA asume que los hilos CUDA se ejecutan en una unidad física distinta, la cual actúa como co-procesador (device) al procesador (host). Como CUDA C es una extensión del lenguaje de programación C, permite al programador definir funciones C, llamadas kernels, las cuales al ser invocadas son ejecutadas en paralelo por N hilos diferentes en la GPU.

Los kernels son el componente principal del modelo de programación de CUDA, son funciones invocadas desde el host (CPU Central) y ejecutadas en el dispositivo. Cuando se invoca un kernel, éste se ejecuta N veces en N hilos diferentes. Cada hilo se diferencia de los demás por su identificador, el cual es único y accesible en el kernel a través de una variable interna y predefinida de CUDA llamada `threadIdx`. A través de `threadIdx` se puede definir el comportamiento específico de cada uno de los hilos.

Para la definición de un kernel se deben respetar varias condiciones, las cuales son:

- El tipo de la función kernel es `void`.
- Debe llevar la etiqueta `__global__`, la cual identifica a un kernel y determina que la función es invocada desde el host (CPU) y ejecutada en el device (GPU).
- Todos los hilos que se activen durante la ejecución del kernel, ejecutan el mismo programa, el cual coincide con el kernel que lo activó.
- El número de hilos es conocido antes de la ejecución del kernel, ellos serán agrupados, según se indica en la invocación, en grupos denominados bloques. Todos los bloques tienen igual número de hilos.

Existe una jerarquía perfectamente definida sobre los hilos de CUDA. Los hilos se agrupan en bloques, los cuales se pueden ver como vectores (una dimensión) o matrices (dos o tres dimensiones). Los hilos de un mismo bloque pueden cooperar entre sí, compartiendo datos y sincronizando sus ejecuciones. Sin embargo, los hilos de distintos bloques no pueden cooperar entre sí.

Los bloques a su vez, se organizan en una malla, la cual puede ser de una o dos dimensiones (en las nuevas arquitecturas, se admiten tres dimensiones). Los bloques e hilos por bloque que tendrá una malla son valores establecidos antes de la invocación, los cuales permanecen invariables durante toda la ejecución del kernel. Dada la organización que provee CUDA para los hilos y como cada uno de ellos tiene un identificador único (`threadIdx`).

Más específicamente `threadIdx` tiene tres componentes (`x, y, z`), permitiendo según la dimensión del bloque, identificar con precisión cada hilo. Cuando el bloque es de una dimensión, las componentes `y` y `z` tienen el valor 1, en el caso de un bloque de dos dimensiones sólo la componente `z` tiene el valor 1.

Lo mismo ocurre con los bloques y las mallas, pero para ellos CUDA tiene definida tres variables: `blockIdx` y `blockDim` para bloques, y `gridDim` para mallas, todas de tres componentes. `blockIdx` permite identificar a los bloques y las variables `blockDim` y `gridDim` contienen el tamaño de cada bloque y de cada malla, respectivamente.

Un programa CUDA está compuesto de una o más fases, las cuales son ejecutadas en el host o en el dispositivo. Aquellas partes que exhiben poco o nada de paralelismo se implementan en el código a ejecutar sobre el host, no así las que pueden ser resueltas aplicando paralelismo de datos, éstas son implementadas a través de código que se ejecutará en el dispositivo, en este caso la GPU. Si bien en el programa CUDA existen dos partes bien diferenciadas, será el compilador el responsable de su diferenciación.

Para ello, el código desarrollado para ejecutarse en el host será compilado con el compilador estándar de C (o el del lenguaje secuencial utilizado) y ejecutado en la CPU como un proceso común. El código a ejecutarse en el dispositivo, escrito en C extendido con palabras claves que expresan el paralelismo de datos y las estructuras de datos asociadas, será compilado con el compilador propio de CUDA (nvcc) (Kandort Edward, 2010).

La Figura 2.4 muestra de manera gráfica la ejecución de un programa en CUDA.

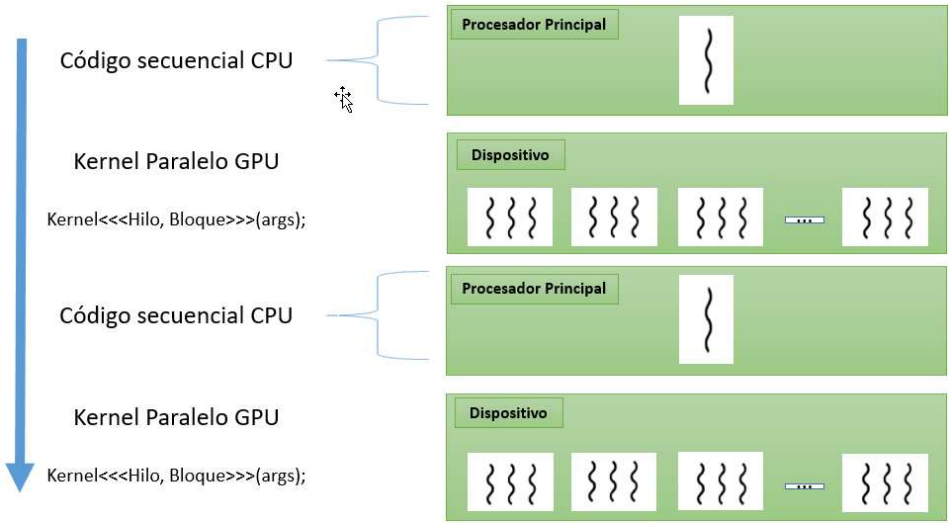


Figura 2.4: Ejecución de un programa CUDA

2.3 MODELO DE PROGRAMACIÓN CUDA

CUDA propone un modelo de programación SIMD (Instrucción Simple-Múltiples Datos) con funcionalidades de procesamiento de vector. La programación de GPU se realiza a través de una extensión del lenguaje estándar C/C++ con constructores y palabras claves. La extensión incluye dos características principales: la organización del trabajo paralelo a través de hilos concurrentes y la jerarquía de memoria de la GPU con sus diferentes costos de acceso. Los hilos en el modelo CUDA son agrupados en Bloques, los cuales se caracterizan por:

- El tamaño del bloque: Cantidad de hilos que lo componen determinada por el programador.
- Todos los hilos de un bloque se ejecutan sobre el mismo SM (Streaming Multiprocessor).
- Los hilos de un bloque comparten la memoria, la cual pueden usar como medio de comunicación entre ellos.

Varios bloques forman una malla y los hilos de diferentes bloques de un malla no se pueden comunicar entre si, esto permite que el administrador de bloques sea rápido y flexible, no tiene en cuenta el número de SM utilizados para la ejecución del programa. Además de las variables en la memoria compartida, los hilos tienen acceso a otros dos tipos de variables: locales y globales. Las variables locales residen en la memoria dinámica de acceso aleatorio (DRAM) de la tarjeta y son privadas a cada hilo.

Las variables globales también residen en la memoria DRAM de la tarjeta, se diferencian de las locales en que pueden ser accedidas por todos los hilos aunque pertenezcan a distintos bloques. Esto lleva a una manera de sincronización global de los hilos.

Como la memoria DRAM es más lenta que la memoria compartida, los hilos de un bloque se pueden sincronizar mediante una instrucción especial, la cual es implementada en memoria compartida.

En un programa CUDA se diferencian dos ámbitos: el host y el dispositivo (device). El host es el ordenador al cual está conectada la tarjeta gráfica y será quien rija su comportamiento. El device es la tarjeta gráfica.

La comunicación de datos entre el host y el dispositivo se lleva a cabo a través de la memoria, sin embargo cada uno (el host y el device) tiene su propio espacio de memoria, las cuales son independientes.

Para resolver un problema en la GPU, se necesita transferir los datos de entrada del programa a la GPU y una vez obtenidos los resultados, transferirlos a la CPU.

CUDA proporciona funciones para realizar estas tareas las cuales se muestran a continuación:

Listado 1: Directivas CUDA para la transferencia entre el host y el device

```
//Copia la variable dev_a del host al device
Memcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);

//Devuelve el resultado desde la GPU a la variable C del host
Memcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);
```

La función **cudaMemcpyHostToDevice** copia de la memoria principal a la memoria del dispositivo y la función **cudaMemcpyDeviceToHost** copia desde la memoria del dispositivo a la memoria principal del host.

Sin embargo para poder realizar las transferencias a memoria es necesario gestionar la memoria global en la GPU, por lo que CUDA proporciona funciones para asignar y liberar espacio de memoria. A continuación se muestran dichas funciones:

Listado 2: Directivas CUDA para la asignación y la liberación de memoria

```
//Reserva memoria en la GPU
cudaMalloc( (void**)&dev_a, N * sizeof(int));

//Libera la memoria reservada de la GPU
cudaFree( dev_a );
```

La función `cudaFree`, libera el espacio de memoria apuntado por la variable que recibe como parámetro.

Una vez asignada la memoria en el dispositivo a cada uno de los objetos con los que se va trabajar, es necesario transferir los datos desde el host al device.

En CUDA, la función kernel especifica el código a ser ejecutado por todos los hilos en forma paralela en el dispositivo. Como los hilos ejecutan el mismo código sobre distintos conjuntos de datos, el código es un ejemplo del modelo SIMD.

2.4 OPENMP

La API OpenMP fue desarrollada para permitir la programación paralela de memoria compartida portátil. Su objetivo es apoyar la paralelización de aplicaciones en varias disciplinas. Además, sus creadores intentaron proporcionar un enfoque que fue relativamente fácil de aprender y aplicar. También se consideró permitir a los programadores trabajar con un único código fuente es decir un único conjunto de

archivos fuente que contiene el código para las versiones secuencial y paralela de un programa, por lo que el mantenimiento del programa se simplifica mucho.

OpenMP está compuesto de tres elementos:

- Directivas de compilación.
- Rutinas de librería.
- Variables de entorno.

Estos objetivos han contribuido en gran medida a darle a la API de OpenMP su forma actual.

OpenMP se basa en un gran cuerpo de trabajo que admite la especificación de programas para su ejecución por una colección de hilos que cooperan entre sí. El sistema operativo crea un proceso para ejecutar un programa y asigna recursos a ese proceso, incluidas páginas de memoria y registros para almacenar valores de objetos. Si varios hilos colaboran para ejecutar un programa, compartirán los recursos, incluido el espacio de direcciones.

Los hilos individuales necesitan solo unos pocos recursos propios: un contador de programa y un área en la memoria para guardar las variables que son específicas para él (incluidos los registros y una pila). Se pueden ejecutar múltiples hilos en un solo procesador o núcleo y se pueden intercalar a través de multihilado simultáneo (multithreading).

Algunas ventajas de OpenMP son las siguientes:

- Estandarización: Proporciona un estándar entre la variedad de arquitecturas de memoria compartida.
- Es claro y directo: Establece un conjunto de simples y limitadas directivas para equipos donde se puede utilizar memoria compartida. Se puede implementar un paralelismo significativo utilizando solo tres o cuatro directivas.
- Fácil de usar: Proporciona la capacidad de poner en paralelo de forma incremental un programa secuencial, a diferencia de librerías de paso de mensajes que requieren de un enfoque de todo o nada. Proporciona la capacidad de implementar paralelismo con granularidad gruesa y granularidad fina.
- Portabilidad: Es compatible con C/C++ y Fortran. Puede ser utilizada en sistemas operativos como Windows, Unix y Linux.

2.5 MODELO DE PROGRAMACIÓN OPENMP

Así como existen diferentes clases de hardware paralelo también existen diferentes modelos para la programación paralela. OpenMP se centra en el modelo de memoria compartida (o espacio de direcciones compartido) como modelo de programación. Este modelo como su nombre lo indica asume que los programas serán ejecutados en más de un procesador donde compartirán la memoria del equipo (Chapman Barbara, 2008).

OpenMP se basa en directivas: el paralelismo se especifica a través de directivas que se insertan el código.

Los hilos individuales necesitan solo un par de recursos para sí mismos: un contador de programa y un área en memoria para salvar variables que son específicos para el hilo incluyendo registros y una pila. Múltiples hilos se pueden ejecutar en un solo procesador o núcleo a través de cambio de contexto que puede intercalarse mediante multihilado simultaneo.

Los hilos se ejecutan simultáneamente en varios procesadores o núcleos que pueden trabajar concurrentemente para ejecutar un programa en paralelo. Programas multihilados pueden ser escritos de diferentes maneras algunas de las cuales permiten interacciones complejas entre hilos.

OpenMP intenta facilitar la programación y ayudar al usuario a evitar una serie de problemas potenciales de programación ofreciendo un enfoque estructurado para la programación multihilo. OpenMP es compatible con el modelo de programación "fork-join", como se muestra en la Figura 2.5.

Bajo este enfoque el programa comienza con un único hilo de ejecución. El hilo que ejecuta este código se le conoce como hilo maestro. Cada vez que un constructor paralelo de OpenMP se encuentra con un hilo mientras se está ejecutando el programa, se crea un conjunto de hilos (el "fork") y se convierte en el hilo maestro del conjunto de hilos y colabora con los demás hilos del conjunto para ejecutar el código dinámicamente encapsulando al constructor.

Al final del código del constructor, solo el hilo original o el hilo maestro del conjunto de hilos continua y los demás finalizan (el "join"). Cada parte del código encapsulado por el constructor paralelo es llamada región paralela.

El trabajo de la implementación de OpenMP en un código es separar las secciones de bajo nivel y crear hilos independientes para ejecutar el código y asignarles trabajo, esto dependiendo de la estrategia que sea requerida.

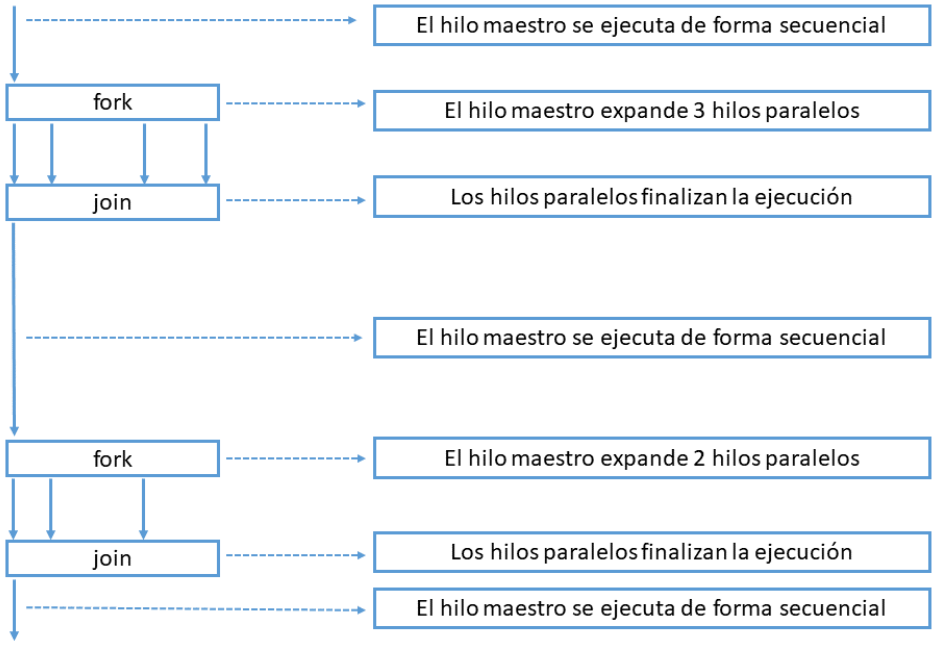


Figura 2.5: Modelo fork-join de memoria compartida.

2.5.1 *Características de OpenMP*

OpenMP comprende un conjunto de directivas de compilación, rutinas de biblioteca de tiempo de ejecución, y las variables de entorno para especificar el paralelismo de memoria compartida en los lenguajes FORTRAN y C/C++.

OpenMP es comúnmente utilizado para incrementar la paralización en un código secuencial existente, y esta tarea se logra mediante la creación de regiones paralelas. Para lograr esto, se especifica la región paralela insertando una directiva antes del código que se va a ejecutar en paralelo para indicarle al compilador que deberá a empezar a ejecutar la sección en paralelo.

Una directiva en OpenMP es una instrucción con un formato especial utilizando la palabra "#PRAGMA"que se aplica generalmente a una parte del código seguido por el código secuencial.

Una rutina o una directiva de OpenMP afectan solo a aquellos que se encuentran contenidas en estas. Muchas de las directivas se aplican a un bloque estructurado del código y una sentencia ejecutable en C/C++, puede ser un conjunto de declaraciones con una sola entrada y una sola salida. En otras palabras, el programa no puede ramificarse adentro o afuera de los bloques del código asociados con las directivas. El final del bloque es explícito en C/C++, solo el comienzo debe ser marcado.

OpenMP proporciona los medios para que el programador pueda:

- **Crear grupos de hilos para la ejecución paralela.**
- **Especificar como repartir el trabajo entre el grupo de hilos.**
- **Declarar variables compartidas y privadas.**
- **Sincronizar los hilos y habilitarlos para poder ejecutar solo ciertas operaciones (es decir, sin la interferencia de los otros hilos).**

Al final de la región paralela existe una barrera implícita de sincronización: esto significa que ningún hilo puede avanzar hasta que todos los otros hilos en el grupo han alcanzado ese punto del programa. Posteriormente, la ejecución del programa continúa con el hilo o los hilos que existían previamente.

Si un conjunto de hilos ejecutan una región paralela y se encuentra con otra directiva paralela, cada hilo en el conjunto actual crea un conjunto de nuevos hilos y se convierten en hilos maestros de cada nuevo conjunto. La anidación permite la realización de programas paralelos de varios niveles.

Las directivas OpenMP siguen las convenciones de los estándares para directivas de compilación en C/C++, son case sensitive, solo puede especificarse un nombre de directiva por directiva y cada directiva se aplica, al menos, a la sentencia que la sigue. que puede ser un bloque estructurado. En directivas largas puede continuarse en la siguiente línea haciendo uso del carácter al final de la línea.

A continuación se muestra como se construyen regiones paralelas en OpenMP.

Listado 3: Directivas en OpenMP

```
//Inicio de region paralela
#pragma omp parallel private(identificador)
{

//Obtiene el numero identificador de cada hilo
identificador = omp_get_thread_num();

//Obtiene el numero de hilos que se levantaron
nhilos = omp_get_num_threads();
}
//fin de region paralela
```

Existen clausulas en las directivas que permiten dirigir el comportamiento de la directiva. En particular, pueden utilizarse para especificar paralelizaciones condicionales, especificar el grado de concurrencia (número de hilos) e incluyen mecanismos para la gestión de los datos. Algunas de las cláusulas aplicables a directivas OpenMP son:

- Paralelización condicionada: Sólo puede utilizarse una clausula if en una directiva paralela.
- Grado de concurrencia: Clausulas como *num_threads, nowait, ordered.*
- Gestión de datos: Dentro de esta categoría se tienen directivas como *private, firstprovate, lastprivate, copyin* entre otras.

El estándar OpenMP define una API para llamadas a funciones de librería. Así, encontramos funciones para averiguar el número de hilos y procesos, y para establecer el número de hilos a utilizar, funciones de propósito general que permiten la creación y gestión de semáforos, funciones para la temporización y medición de tiempos, y funciones para paralelismo anidado y para la gestión dinámica de hilos. Algunas de las funciones se mencionan a continuación:

- void omp_set_num_threads: Establece el número de hilos a utilizar en las siguientes regiones paralelas.
- int omp_get_num_threads(void): Devuelve el número de hilos que se encuentran actualmente en el equipo ejecutando la región paralela desde la que se invoca.
- int omp_get_max_threads(void): Devuelve el máximo valor que puede ser devuelto por una llamada a la función del punto anterior.
- int omp_get_thread_num(void): Devuelve el número de hilo asignado dentro de la llamada en el conjunto de hilos.
- int omp_get_num_procs(void): Devuelve el número de procesadores físicos disponibles por el programa.

3

MODELACIÓN NUMÉRICA DEL PROBLEMA DIRECTO DE LA GRADIOMETRÍA GRAVIMÉTRICA

Se aborda el algoritmo computacional para calcular el tensor y el gradiente del tensor gravimétrico a partir de un ensamble de prismas.

3.1 INTRODUCCIÓN

Desde los primeros fundamentos teóricos en los siglos XVII Y XVIII, hasta las actuales medidas de la gravedad, el método gravimétrico o bien la determinación de la variación del campo gravitatorio, ha estado guiada por la interacción entre las posibilidades tecnológicas y los objetivos científicos.

Una de las características peculiares en el desarrollo histórico de la gravimetría en los últimos 300 años, ha sido el continuo crecimiento de la cobertura gravimétrica en los continentes y océanos, así como la mejora en la exactitud de las medidas. La gravedad es usada en los estudios de los procesos dinámicos dentro de la Tierra, y también es importante en la Geofísica de exploración.

3.2 GRAVIMETRÍA

La gravimetría es la parte de la Geofísica que mide las variaciones laterales de la atracción hacia el centro de la tierra (Ramos, 2008).

La gravimetría se basa en el estudio del campo gravimétrico terrestre con el fin de detectar cambios de materiales o variaciones en la densidad de los mismos. La propiedad física de las rocas que aprovecha el método gravimétrico es la densidad. Un cambio en la densidad de las rocas del subsuelo trae como consecuencia un cambio en el campo potencial gravimétrico terrestre.

Se aplica principalmente en trabajos de reconocimiento que, utilizados con información directa del subsuelo, dan como resultado, planos confiables a menudo usados en Geología de minas o petrolera. La prospección gravimétrica suele realizarse en forma de malla de tal manera que se puede definir un mapa 2D de gravedad, resultado de aplicar distintas correcciones a los datos originales.

Las mediciones para la detección de las variaciones de densidad de la tierra se realizan a través de un gradiometro, el cual identifica sobre una superficie variaciones, encontrando así los cuerpos de baja densidad.

Por ejemplo, un avión equipado que sobrevuela una montaña o un cuerpo salino subterráneo de baja densidad (un déficit de masa) mostrará solo variaciones sutiles de la atracción gravitatoria global. En cambio, la medición de los correspondientes gradientes gravitatorios revelará esos accidentes geológicos inmediatamente. El movimiento errático del avión crea un ruido considerable en un solo perfil gravitatorio, mientras que la medida de la diferencia entre dos sensores para obtener el gradiente elimina automáticamente la fuente del error como se muestra en la Figura 3.6

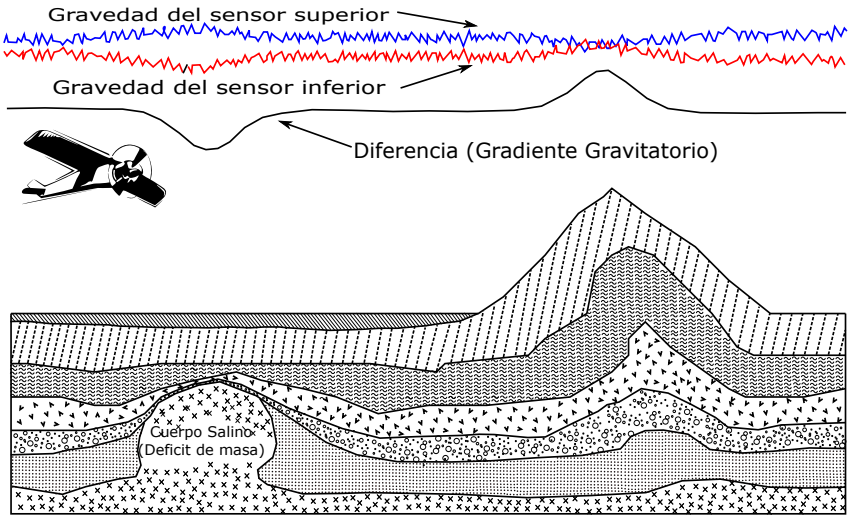


Figura 3.6: Cartografía de los gradientes gravitatorios.

También se puede utilizar para la elaboración de mapas estructurales profundos, aunque a veces es afectada por los contrastes someros de densidad, otras aplicaciones se dan en la geotecnia (detección de huecos y cavidades), estudios de la corteza y fuentes geotermales.

3.3 CÁLCULO DEL TENSOR Y GRADIENTE DEL TENSOR GRAVIMÉTRICO

Modelar la aceleración gravitacional es una herramienta común en la Geofísica para probar modelos de la distribución de densidad dentro de la superficie. A menudo la información tectónica o los modelos sísmicos se utilizan para definir estructuras

geológicas amplias con una densidad común y éstas se parametrizan como cuerpos poligonales dentro del esquema de modelado numérico.

La fuerza de gravedad se describe con tres componentes en las direcciones x, y y z. Sin embargo el gradiente, necesita nueve componentes, que forman una matriz, y representa el campo gravitatorio en forma tensorial.

Este tipo de enfoque tiene la ventaja de que el número de cuerpos se mantiene bajo, incluso para modelos complejos, lo que facilita la construcción del modelo y reduce el número de veces que se evalúan las funciones. Para poder reducir el tiempo del cálculo del tensor del gradiente gravimétrico es posible emplear tarjetas gráficas para acelerar el trabajo de cómputo haciendo uso de mayor ancho de banda y memoria que los CPUs.

En 1687, Newton publico la ley de atracción gravitacional que dice: La magnitud de la fuerza gravitacional entre dos masas es proporcional a cada masa e inversamente proporcional al cuadrado de su separación. En coordenadas cartesianas la fuerza mutua entre una partícula de masa m centrada en el punto $\mathbf{Q} = (\mathbf{x}', \mathbf{y}', \mathbf{z}')$ y una partícula de masa m en P = (x, y, z) es dado por:

$$U(P) = \gamma \frac{m}{r} \tag{1}$$

Donde r es igual a la distancia entre el punto Q y el punto P:

$$\mathbf{r} = [(\mathbf{x} - \mathbf{x}')^2 + (\mathbf{y} - \mathbf{y}')^2 + (\mathbf{z} - \mathbf{z}')^2]^{\frac{1}{2}} \tag{2}$$

Y donde γ es la constante gravitacional de Newton que es equivalente a:

$$\gamma = 6.67428x10^{-11} \frac{m^3}{s^2 kg} \tag{3}$$

La función U se llama el potencial gravitacional o potencial de Newton. A partir de campo escalar U, se obtiene un campo vectorial g como el gradiente U, por lo tanto, se puede expresar como:

$$g = \nabla U \tag{4}$$

Donde ∇ expresándolo en coordenadas cartesianas tenemos que:

$$\nabla = \frac{\partial}{\partial x}\mathbf{i} + \frac{\partial}{\partial y}\mathbf{j} + \frac{\partial}{\partial z}\mathbf{k} \tag{5}$$

Los componentes vectoriales del campo $\mathbf{g}$ se definen en términos del campo potencial como:

$$\mathbf{g_x} = \frac{\partial \mathbf{U}}{\partial \mathbf{x}}, \mathbf{g_y} = \frac{\partial \mathbf{U}}{\partial \mathbf{y}}, \mathbf{g_z} = \frac{\partial \mathbf{U}}{\partial \mathbf{z}} \tag{6}$$

El campo gravitatorio se puede expresar en términos del potencial de gravedad $U_g(r)$ como:

$$g(r) = \nabla U_g(r) \tag{7}$$

El potencial gravitacional U de un punto ρ debido a una masa con densidad en una región con volumen V se expresa de la siguiente manera:

$$U(P) = \gamma \int\int\int_V \frac{\rho}{|r - r'|} dv, \tag{8}$$

Donde γ es la constante universal de gravedad ρ es la densidad en el volumen, r es la posición del punto de observación y $\mathbf{x}'$ es el elemento del volumen dado por dv.

La gravedad debida a un cuerpo tridimensional con densidad $\jmath(\mathbf{x}', \mathbf{y}', \mathbf{z}')$ y un punto de observación arbitrario en $\rho(x_0, y_0, z_0)$ es:

$$g(r)_\alpha = \gamma \int\int\int_V \rho(x', y', z') \left(\frac{\alpha - \alpha'}{\sqrt{(x - x')^2 + (y - y')^2 + (z - z')^2}} \right) dv, \tag{9}$$

En donde α representa las diferentes direcciones en x,y,z y $\mathbf{dv} = (\mathbf{dx'}, \mathbf{dy'}, \mathbf{dz'})$ por lo tanto $g(r)_\alpha$ se puede escribir como:

$$g(r)_x = -\gamma \int\int\int_V \rho(x', y', z') \frac{x - x'}{\sqrt{(x-x')^2 + (y-y')^2 + (z-z')^2}^{\frac{3}{2}}} dv, \quad (10)$$

$$g(r)_y = -\gamma \int\int\int_V \rho(x', y', z') \frac{y - y'}{\sqrt{(x-x')^2 + (y-y')^2 + (z-z')^2}^{\frac{3}{2}}} dv, \quad (11)$$

$$g(r)_z = -\gamma \int\int\int_V \rho(x', y', z') \frac{z - z'}{\sqrt{(x-x')^2 + (y-y')^2 + (z-z')^2}^{\frac{3}{2}}} dv, \quad (12)$$

Para calcular los tensores aplicamos la segunda derivada parcial del potencial gravitacional:

$$g_{\alpha\beta}(r) = \frac{\partial^2}{\partial_\alpha \partial_\beta}, \alpha, \beta = x, y, z \quad (13)$$

Aplicando la segunda derivada a los componentes vectoriales respecto a x, y, z tenemos que:

$$g_{xx}(r) = \gamma \int\int\int_V \frac{3(x-x')^2 - (r-r')^2}{\mid r - r' \mid^5} \rho(r') dv, \quad (14)$$

$$g_{yy}(r) = \gamma \int\int\int_V \frac{3(y-y')^2 - (r-r')^2}{\mid r - r' \mid^5} \rho(r') dv, \quad (15)$$

$$g_{zz}(r) = \gamma \int\int\int_V \frac{3(z-z')^2 - (r-r')^2}{\mid r - r' \mid^5} \rho(r') dv, \quad (16)$$

$$g_{xy}(r) = \gamma \int\int\int_V \frac{3(x-x')(y-y')}{|\, r-r' \,|^5}\rho(r')dv, \tag{17}$$

$$g_{xz}(r) = \gamma \int\int\int_V \frac{3(x-x')(z-z')}{|\, r-r' \,|^5}\rho(r')dv, \tag{18}$$

$$g_{yz}(r) = \gamma \int\int\int_V \frac{3(y-y')(z-z')}{|\, r-r' \,|^5}\rho(r')dv, \tag{19}$$

Para poder simplificar la notación del tensor gravitacional podemos escribir cada uno de sus componentes en forma matricial de la siguiente manera (Moorkamp et al., 2010b):

$$\mathbf{r} = \begin{bmatrix} g_{xx} & g_{xy} & g_{xz} \\ g_{yx} & g_{yy} & g_{yz} \\ g_{zx} & g_{zy} & g_{zz} \end{bmatrix}, \tag{20}$$

La ecuación para el efecto de un solo prisma de densidad ρ sobre la aceleración gravitatoria vertical g_z es (Li Xiong, 1997).

$$g_z = -\gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk}(x_i \ln(y_j + r_{ijk}) + y_j \ln(x_i + r_{ijk}) + z_k \arctan\frac{z_k r_{ijk}}{x_i y_j}) \tag{21}$$

Y para el caso en que tenemos dos elementos del tensor gravimétrico es:

$$g_{xx} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \arctan\frac{y_j z_k}{x_i r_{ijk}}, \tag{22}$$

$$g_{xy} = -\gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \ln(z_k + r_{ijk}), \tag{23}$$

Donde:

$$x_i = x - \xi_i y_j = y - \eta_j z_k = z - \zeta_k$$

$$r_{ijk} = \sqrt{x_i^2 + y_j^2 + z_k^2}$$

$$\mu_{ijk} = (-1)^i (-1)^j (-1)^k$$

Podemos calcular todos los demás elementos del tensor de gravimetría mediante la permutación de los ejes de coordenadas (e.g. Li and Chouteau, 1998; Nagy et al., 2000). Además, el tensor es simétrico así que solo tenemos que calcular 6 en lugar de los 9 elementos del tensor. Teóricamente tenemos solo que calcular 5 elementos, ya que los términos diagonales del tensor están relacionados por la ecuación de Poisson.

$$\frac{\partial^2 g}{\partial x^2} + \frac{\partial^2 g}{\partial y^2} + \frac{\partial^2 g}{\partial z^2} = -4\pi\gamma\rho \tag{24}$$

Para las demás componentes se tiene que:

$$g_{zz} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \arctan\frac{x_i y_j}{z_k r_{ijk}}, \tag{25}$$

$$g_{yy} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \arctan\frac{x_i z_k}{y_j r_{ijk}}, \tag{26}$$

$$g_{xz} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \ln(y_j + r_{ijk}), \tag{27}$$

$$g_{yz} = \gamma\rho \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k=1}^{2} \mu_{ijk} \ln(x_i + r_{ijk}), \tag{28}$$

El elemento g_{xy}, por ejemplo, es una medida de velocidad con el que el componente de la gravedad en la dirección x (g_x) cambia al moverse uno por la dirección de y. El elemento g_{yz} representa el cambio de la componente de la gravedad en la dirección y (g_y) al moverse por la dirección z, y así sucesivamente. Las componentes del vector se muestran en la Figura 3.7

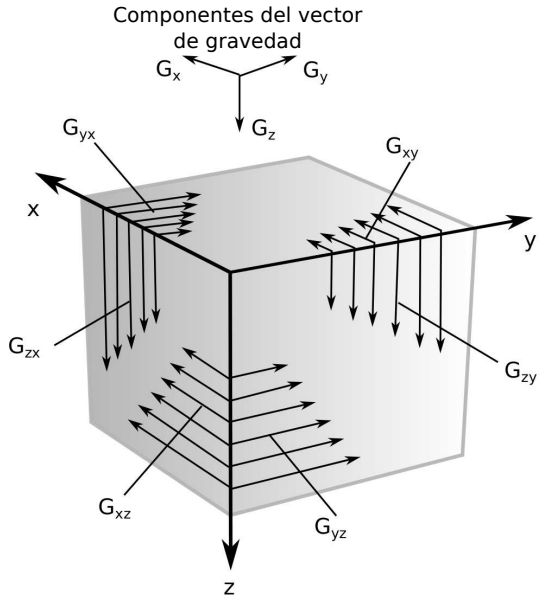


Figura 3.7: Componentes del tensor gravimetrico.

4

DISEÑO E IMPLEMENTACIÓN DEL ALGORITMO EN OPENMP Y CUDA

En este capítulo se presenta el diseño e implementación del algoritmo numérico utilizando OpenMP y CUDA utilizado para calcular el tensor del gradiente gravimétrico a partir de un ensamble de prismas rectangulares.

4.1 TRABAJOS RELACIONADOS

En trabajos previos se han presentado implementaciones para calcular el tensor del gradiente gravimétrico utilizando las arquitecturas CUDA y MPI (Couder-Castaneda et al., 2013) en el cual se realizó una descomposición del problema en subconjuntos de prismas distribuidos en procesos MPI en donde cada proceso MPI controla una tarjeta NVIDIA.

Sin embargo el uso de la tecnología MPI utilizada tiene algunas desventajas, desde que la instalación y configuración del cluster así como las dificultades que la programación distribuida implica, ya que al utilizar MPI implica que los procesos son disjuntos, es decir, operan como entidades distintas y la región de memoria en la que se realizan es privada para cada proceso lo que complica una comunicación entre GPUs ya que operan en asignaciones de memoria diferentes.

4.2 DISEÑO

En este trabajo se decidió utilizar la tecnología OpenMP en lugar de MPI en el diseño del algoritmo para calcular el tensor del gradiente gravimétrico.

La metodología de diseño que se utilizó pretende un enfoque, donde los problemas relacionados con la concurrencia son tomados en cuenta desde un principio. Esta metodología de diseño consta de cuatro etapas: partición, comunicación, aglomeración y mapeo como se muestra en la Figura 4.8, las etapas se describen continuación:

- Partición: El cómputo se diseñará contemplando la máxima granularidad posible sobre los datos operados, por lo que el cómputo se debe descomponer en tareas pequeñas. Se identifican las zonas del programa que pueden ser ejecutadas de forma paralela.
- Comunicación: Se determina la comunicación requerida para coordinar la ejecución de las tareas, y se definen las estructuras de comunicación y los algoritmos apropiados.
- Aglomeración: La estructura de las tareas y comunicación definidas en las dos primeras etapas, se evalúan con respecto a los requisitos de rendimiento y los costos de implementación.
- Mapeo: Cada tarea se asigna a un procesador de una manera que intenta hacer el máximo uso del procesador y minimizar los costos de comunicación.

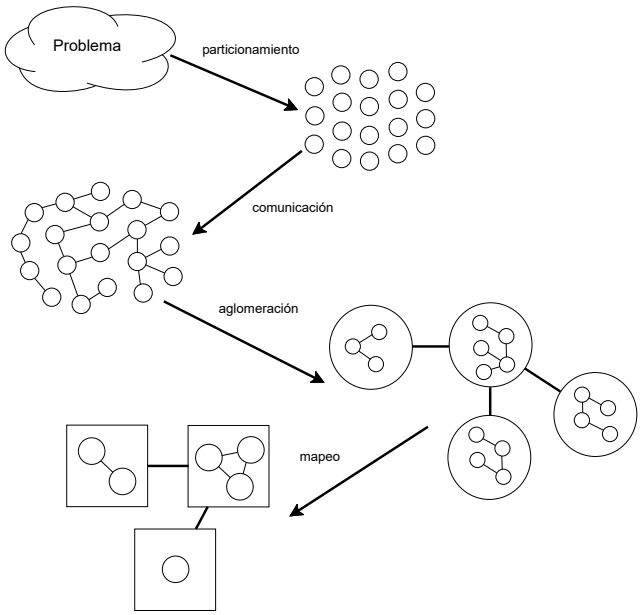


Figura 4.8: Metodología de Foster utilizada para la construcción del algoritmo.

Una de las opciones más simples para la paralelización de este problema utilizando CUDA es particionar por el número de prismas y dividir el dominio de la malla de observación entre el número de núcleos, sin embargo este diseño es muy ineficiente ya que por cada prisma será necesario hacer una llamada hacia el GPU (ejecutar una función kernel).

La aplicación consiste en calcular el gradiente gravimétrico producido por un prisma rectangular con densidad constante con referencia a un conjunto llamado malla de observación como se muestra en la Figura 4.9. El conjunto de prismas es conocido como un ensamble de prismas y no es necesariamente regular.

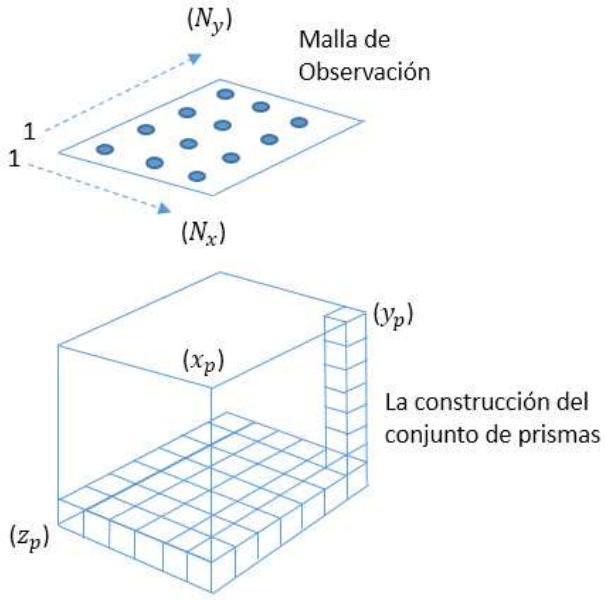


Figura 4.9: Construcción de un prisma de densidades constantes con respecto a una malla de observación.

Como el campo gravitatorio cumple con el principio de superposición, permite descomponer un problema lineal en dos o más subproblemas más sencillos, de tal modo que el problema original se obtiene como la superposición o suma de los problemas más sencillos con respecto a un punto observación como se muestra en la Figura 4.10, si f es la respuesta calculada en un punto (x, y) entonces $f(x, y)$ está dada por:

$$f(x, y) = \sum_{k=1}^{M} G(\rho_k, x, y) \tag{29}$$

donde M es el número total de prismas y p es la densidad del prisma por lo que podemos reescribir la función como:

$$g = f(x_1, y_1, z_1, x_2, y_2, z_2, x, y, z, \rho) \tag{30}$$

donde (x_1, y_1, z_1) es el vértice superior izquierdo y (x_2, y_2, z_2) el vértice inferior derecho del prisma y (x, y, z) es el punto de observación y ρ es la densidad.

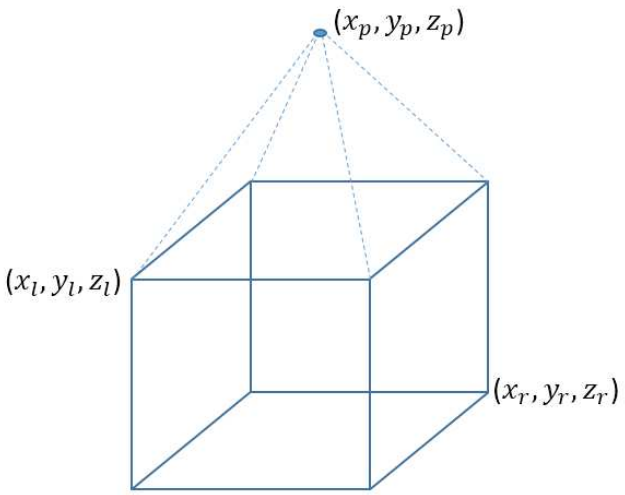


Figura 4.10: Cálculo de una anomalía producida por un prisma con respecto a un punto de observación.

Para poder discretizar el cubo, podemos definir x_ρ,y_ρ, y z_ρ como el número de caras en las direcciones x, y, z, respectivamente. Si el cubo es discretizado de una manera homogénea, entonces podemos definir M como el número de prismas es decir $M = x_p \times y_p \times z_p$ y la numeración consecutiva de los prismas empezaría con x, y y finalmente con z.

En el caso de que el ensamblado de los prismas sea irregular será necesario proporcionar para cada prisma, $x_1, y_1, z_1, x_2, y_2, z_2$ y ρ. Definiremos O como el número de puntos de observación que está determinado como $O = N_x \times N_y$, donde N_x y N_y son el número de puntos de observación en las direcciones x y y respectivamente. Por lo tanto el número de veces que se requiere para llamar a la función definida en (30) para calcular la anomalía producida por un componente es $M \times O$.

De acuerdo a la metología propuesta por Foster (Foster, 1995), el primer paso para desarrollar un programa en paralelo es buscar la granularidad más fina, ya que al determinar la molécula computacional mínima reducimos el riesgo de no considerar alguna opción de paralelización, aunado a que CUDA maneja un paradigma de múltiples hilos muy finos, comparado con un CPU convencional que maneja una

decena de hilos, la granularidad en CUDA es extremadamente fina.

Analizando el requerimiento de cómputo, para este problema básicamente existen dos opciones, considerar dividir el cálculo por número de prismas o por número de observaciones, por lo que se puede paralelizar por prismas o por puntos de observación, es decir, particionar el cómputo sobre el número de elementos M (prismas) o por el número de elementos O (puntos de observación).

Es necesario analizar las dos opciones de paralelización y examinar cuál tiene un mejor rendimiento. Sin embargo, para este problema el número de prismas es mucho mayor que el número de puntos de observación ($M >> O$), luego entonces y considerando que el número de hilos creados en CUDA para lograr un buen rendimiento superará en demasía el número de puntos de observación, la mejor opción de paralelización para este problema es particionando el número de prismas.

Comenzaremos implementando en un solo GPU es decir utilizando una sola tarjeta que consistirá en construir una malla de observación para cada hilo de ejecución que se creará en la GPU. Si creamos 14 bloques y cada bloque con 32 hilos se habrán generado 448 mallas de observación.

En este experimento analizaremos la opción más simple de paralelización, que consiste en dividir la malla de observación en la memoria de la tarjeta para cada prisma como se muestra en la Figura 4.11.

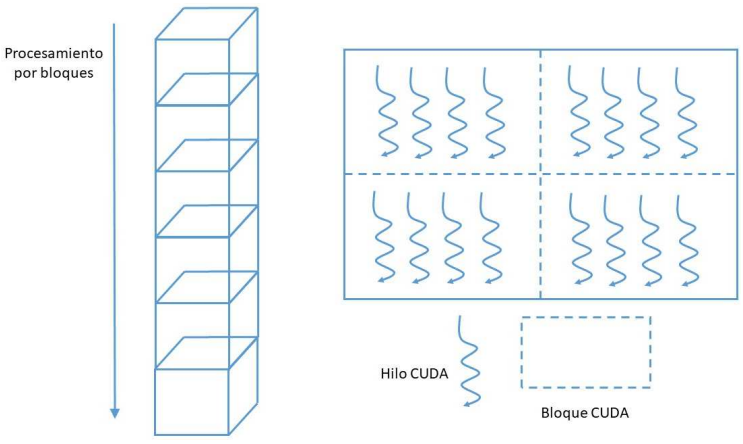


Figura 4.11: Partición por puntos de observación.

En este método, la paralelización se lleva a cabo en el ciclo de las observaciones ésto lo podemos implementar utilizando OpenMP, lo que significa que el compilador

creara la región paralela en el CPU y ejecutará las regiones en la GPU como se muestra en la Figura 4.12.

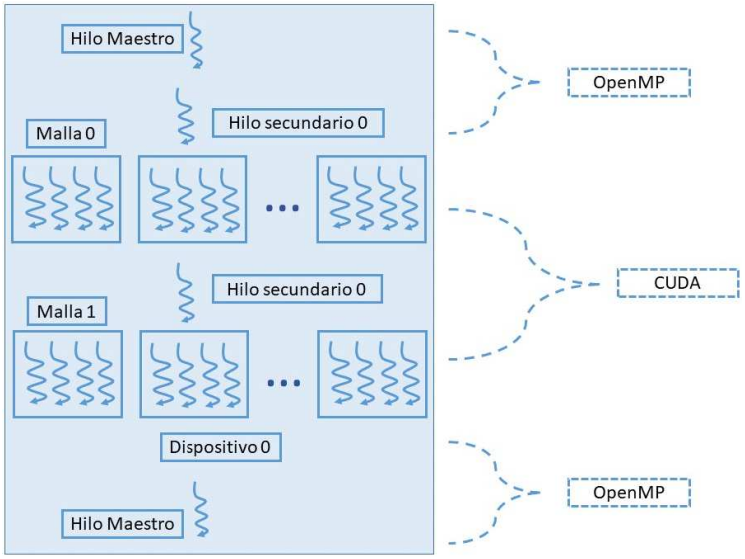


Figura 4.12: Modelo de programación OpenMP/CUDA.

Como se mencionó en el capítulo 2 el modelo de programación de OpenMP se basa en la arquitectura de CPU multinúcleo de memoria compartida y CUDA tiene una arquitectura de memoria compartida de múltiples núcleos GPUs como se puede ver en la Figura 4.13.

La fase serial del programa es ejecutada primero por el hilo maestro en el CPU a continuación, la GPU ejecuta el trabajo en la fase paralela específicamente varios hilos de trabajo de CPU se asignan mediante una instrucción OpenMP y cada hilo de trabajo gestiona un GPU, que se utiliza para ejecutar las funciones paralelas de datos (kernels).

Cuando se inicia un kernel, se genera un gran número de subprocesos del GPU para explotar el paralelismo de datos. Todos esos hilos generados por el kernel realizarán las mismas instrucciones durante la fase paralela.

Después de que la GPU termina su cálculo paralelo, la CPU recupera el tiempo de ejecución y continuará las instrucciones en serie.

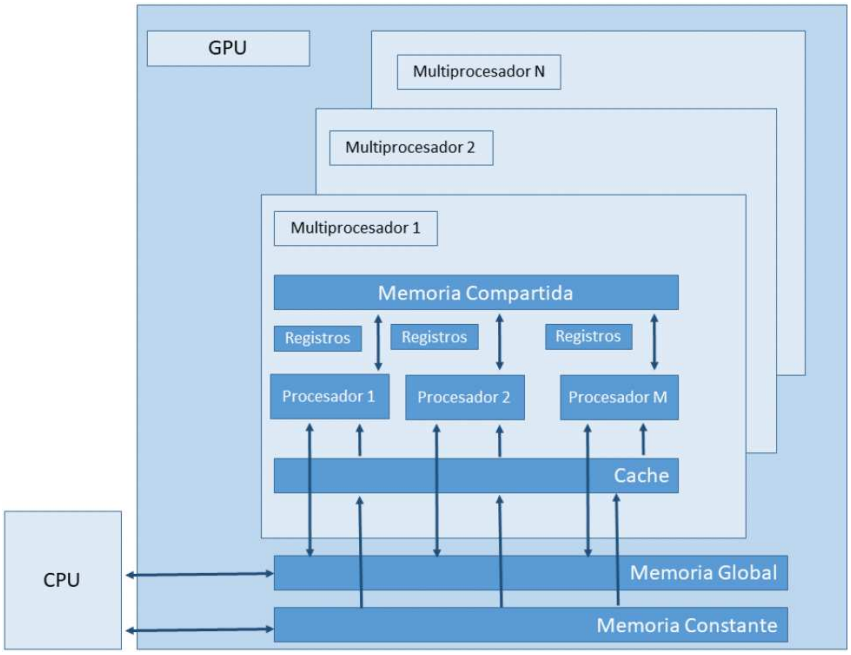


Figura 4.13: Arquitectura GPU CUDA.

Sin embargo este método tiene una gran desventaja debido a que realiza llamadas excesivas a la función del kernel lo que disminuye el rendimiento desde la región paralela ya que en cada llamada a la función se crea y se cierra la región paralela.

Otra opción que tenemos para realizar la paralización es realizar las particiones mediante prismas. Sin embargo, es necesario afrontar los problemas de coherencia, por lo que es necesario crear una malla de observación para cada hilo de ejecución como se puede observar en la Figura 4.14.

Este diseño permite procesar al mismo tiempo tantos prismas como hilos se generen, por ejemplo, si se crean 400 hilos, se procesarán en paralelo el mismo número de prismas al mismo tiempo por cada llamada a la función kernel y así cada vez que se llama a la función.

Tenemos que el número de veces que se llama a la función kernel, está determinada por:

$$p = \frac{M}{T} \tag{31}$$

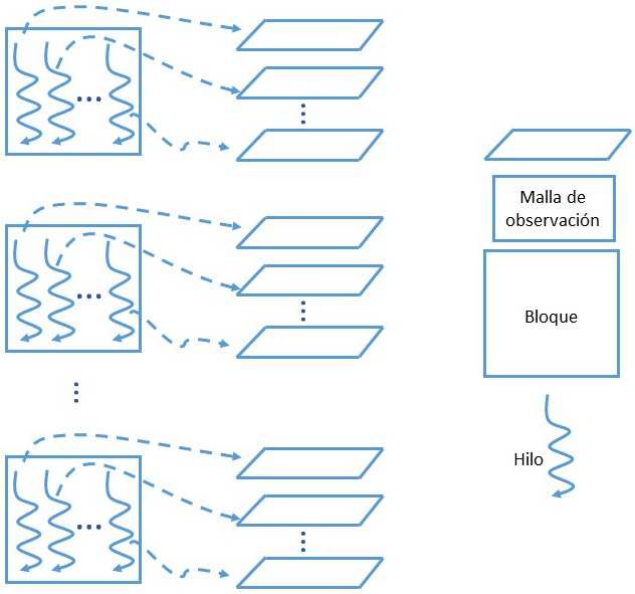


Figura 4.14: Particionado por prismas usando diferentes espacios de memoria.

donde M es el número de prismas y T es el número de hilos creados $p < M$. Para ejemplificar que el modelo de partición mediante prismas es mejor, supongamos que se crean 14 bloques y cada uno de ellos contiene 512 hilos de ejecución por lo que tenemos $T = 14 * 512 = 7168$ mallas de observación, si tenemos un problema con 200,000 mil prismas, el número de llamadas a la función del kernel serán $(200000/7168) = 30$, lo que es un número muy reducido de llamadas hacia la función kernel.

Cada hilo seguirá el esquema de procesamiento sobre los prismas: Un hilo t procesará la secuencia de prismas:

$$T \times (n-1) + t \tag{32}$$

donde $n = 1, 2, 3, \ldots, p$.

La implementación consta de codificar en la GPU las funciones que calculan los componentes vectoriales (g_x,g_y,g_z) y los componentes del tensor (g_{xx},g_{yy},g_{zz},g_{xy}, g_{xz},g_{yz}), las funciones del dispositivo sólo pueden ser llamadas por los kernels y son ejecutadas por un solo hilo CUDA.

Para calcular cualquier componente es necesario asignar la memoria a un arreglo de tres dimensiones G de tamaño $T \times N_y \times N_y$.

Definimos el `ID = thread.x + (block.x * blockDim.x)` donde `thread.x` es el identificador de hilo, `block.x` es el identificador de bloque y `blockDim.x` es el tamaño del bloque. También definimos K como `K = (T)* I + ID`, donde T es el número de hilos creados iguales al número de mallas de observación, I es el número de particiones sobre el conjunto de prismas M en el que está trabajando.

La partición I del conjunto M es una división de M en subconjuntos no superpuestos y no vacíos de tamaño T que cubren todo M. Los subconjuntos son colectivamente exhaustivos y mutuamente excluyentes con respecto al conjunto M. I puede tomar los valores $1, 2, 3, \ldots, p$.

Listado 4: Esquema general del kernel computacional

```
if (K<=M)
{
  For j = 1; Ny; j++
      For i = 1; Nx; i++
        G(Thread,i,j)= Gz(parameters)+G(Thread,i,j);
}
Else
    For j = 1; Ny; j++
    {
      For i = 1; Nx; i++
        G(Thread, i, j)=G(Thread,i,j)+0.0
    }
}
```

Para calcular el resultado final de la anomalía es necesario añadir todos los resultados obtenidos de los hilos, esto es, la anomalía final G_f en un punto (i, j) se aproxima como:

$$G_f(i,j) = \sum_{k=1}^{T} G(k,i,j), \tag{33}$$

Para generar la reducción, podemos hacer uso de OpenMP que genera automáticamente las regiones paralelas y la estructura en C.

Listado 5: Generación automática del kernel utilizando OpenMP

```
#pragma omp parallel (Gd,Gshared d)
{
    for k = 1;T;k++
    {
        #pragma omp for
        for i = 1;Nx;i++
        {
          for j = 1;Ny;j++
            Gshared D(i,j)=Gd(k,i,j) + Gshared D(i,j);
        }
    }
}
```

`Gshared` es la matriz bidimensional donde se realiza la reducción, y `Gd` es la tridimensional que contiene los datos de la anomalía generada por los diferentes hilos.

Con el cálculo de `Gshared` obtenmos los valores finales de la anomalía y por lo tanto es el fin del procedimiento.

5

RESULTADOS EXPERIMENTALES

En este capítulo se llevan a cabo los experimentos de desempeño del diseño propuesto OpenMP+CUDA para Multi-GPU sobre tarjetas TESLA C2070. Los resultados obtenidos en rendimiento son comparables en rendimiento con respecto a un cluster de computadoras.

5.1 CONFIGURACIÓN DEL EXPERIMENTO

El experimento que se utilizó es un caso sintético que está compuesto por un ensamble de $700 \times 700 \times 50$ prismas que conforman 7 esferas de contraste de densidad variable en el subsuelo. Es análogo a imaginar tener siete balones enterrados en la tierra, pero en este caso gigantes, ya que el cubo abarca un volumen de 20 Km $\times$ 20 Km $\times$ 8 Km. La densidad alrededor de las esferas se considera constante por eso se elimina, y las esferas entre más profundas tienen una mayor densidad de tal modo que generen la misma respuesta gradiométrica.

Las esferas están conformadas por 251,946 prismas y una malla de observación de 150 x 100 = 15,000 puntos a una elevación de 100 metros, como se muestra en la Figura 5.15. Por lo tanto, el número de veces que necesitamos invocar la función que calcula un componente del tensor o del vector es de 3,779,190,000 lo que representa un problema de alto costo computacional.

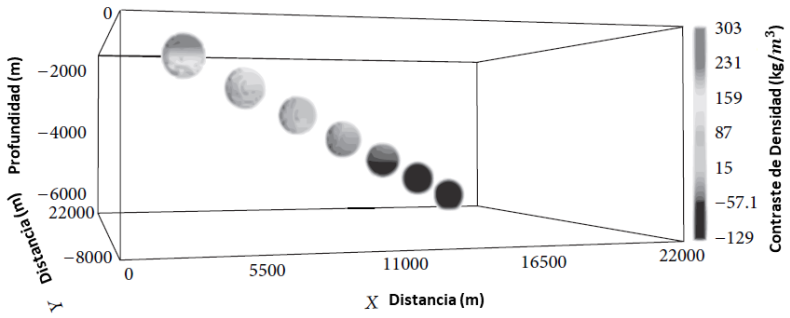
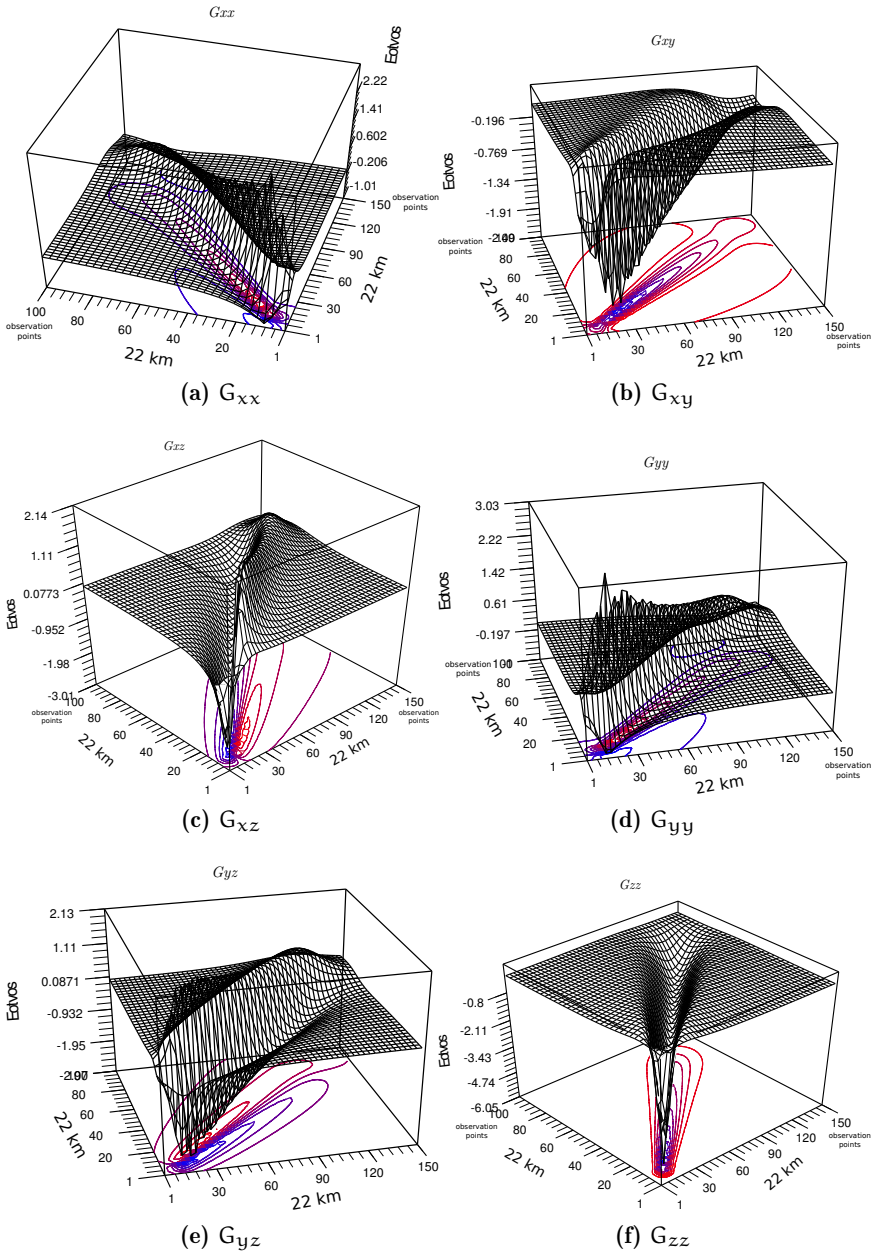


Figura 5.15: Configuración del problema de las 7 esferas de densidad variable en el subsuelo, conformadas por 251,946 prismas.

Se llevaron a cabo experimentos para calcular en conjunto los componentes del componentes del tensor G_{xx}, G_{yy}, G_{zz}, G_{xy}, G_{xz} y G_{yz} en una estación de trabajo con 3 GPUs, mostrados en la Figura 5.16.



(a) G_{xx} **(b)** G_{xy}

(c) G_{xz} **(d)** G_{yy}

(e) G_{yz} **(f)** G_{zz}

Figura 5.16: Respuestas del tensor gravimétrico, obtenidas de las esferas.

Las características de la estación de trabajo donde se llevaron a cabo los experimentos numéricos son las siguientes:

- Sistema operativo Red Hat 6.3.
- 12 Gb de memoria RAM.
- 6 núcleos reales por procesador.
- Tecnología de hiperhilado deshabilitada.
- 3 tarjetas gráficas TESLA modelo C2070.
- 2 procesadores Intel® Xeon® CPU X5690 @3.47 GHz.

Las principales características que podemos resaltar de la tarjeta TESLA C2070 son:

- 515 GLOPS virtuales de doble precisión.
- 14 Multiprocesadores (SM).
- 6 GB de memoria global DDR5.
- 1.15 GHz de frecuencia de los núcleos CUDA.
- 32 núcleos por multiprocesador (448 núcleos CUDA en total).
- 1.03 TFLOPS teóricos en precisión simple.

5.2 EXPERIMENTOS SOBRE UN GPU

El primer experimento se realizó en una tarjeta TESLA C2070 y consiste en realizar pruebas con diferentes tamaños de bloques (número de hilos por bloque), manteniendo el número de bloques fijos en 14. El tamaño del bloque seleccionado obedece al número de multiprocesadores disponibles en la tarjeta (14) y el tamaño del bloque varía en múltiplos de 32 hasta llegar a los 512 hilos por bloque.

En la Tabla 1 se tabulan los resultados obtenidos al ir incrementando el número de hilos en múltiplos de 32 hasta llegar a 512 hilos para la precisión sencilla y doble y la memoria requerida en cada caso.

En la Figura 5.17 se muestran el comportamiento de los los tiempos de ejecución obtenidos para calcular todas las componentes tensoriales y vectoriales producidas por las siete esferas en precisión sencilla y doble tabuladas en la Tabla 1. Se puede apreciar que, al aumentar el tamaño del bloque, el tiempo de ejecución disminuye exponencialmente hasta un límite o un tiempo asintótico que en doble precisión es

de 80 s y en precisión sencilla de 36 s.

Tabla 1: Tiempos de cómputo obtenido, usando precisión sencilla y doble. PS-Precisión Simple, PD-Precisión Doble

Tamaño del bloque	Tiempo PS	Tiempo PD	Memoria PS	Memoria PD
32	442s	804s	87 MB	120 MB
64	225s	413s	112 MB	171 MB
96	152s	290s	138 MB	223 MB
128	116s	225s	163 MB	274 MB
160	95s	193s	215 MB	325 MB
192	81s	145s	189 MB	376 MB
224	75s	132s	240 MB	428 MB
256	73s	115s	266 MB	479 MB
288	51s	109s	292 MB	530 MB
320	48s	99s	317 MB	582 MB
352	44s	95s	343 MB	633 MB
384	41s	88s	363 MB	684 MB
416	41s	88s	394 MB	735 MB
448	39s	82s	420 MB	787 MB
480	39s	82s	445 MB	838 MB
512	36s	80s	471 MB	889 MB

Para obtener una mejor visión de los tiempos de ejecución obtenidos en la Figura 5.17, se calcula el correspondiente speed-up el cual es mostrado en la Figura 5.18. Se puede observar que la reducción del tiempo de cálculo es prácticamente lineal aumentando la ocupación en ambos casos y posteriormente comienza a estabilizarse, lo que significa que obtendremos un incremento en el rendimiento si se aumenta el tamaño del bloque en múltiplos de 32 hilos. Se obtiene el mejor tiempo de cálculo con un bloque de 512 hilos, no obstante, esta es la configuración que consume más memoria.

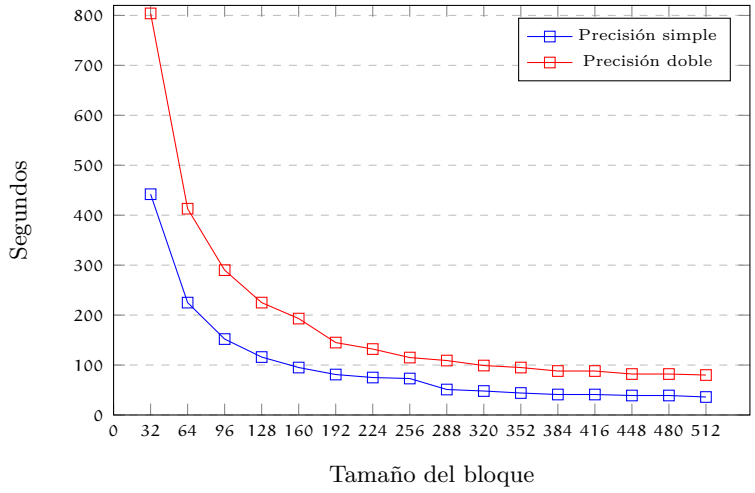


Figura 5.17: Comparación del tiempo de ejecución usando un tamaño de bloque variable en múltiplos de 32, en precisión doble y sencilla.

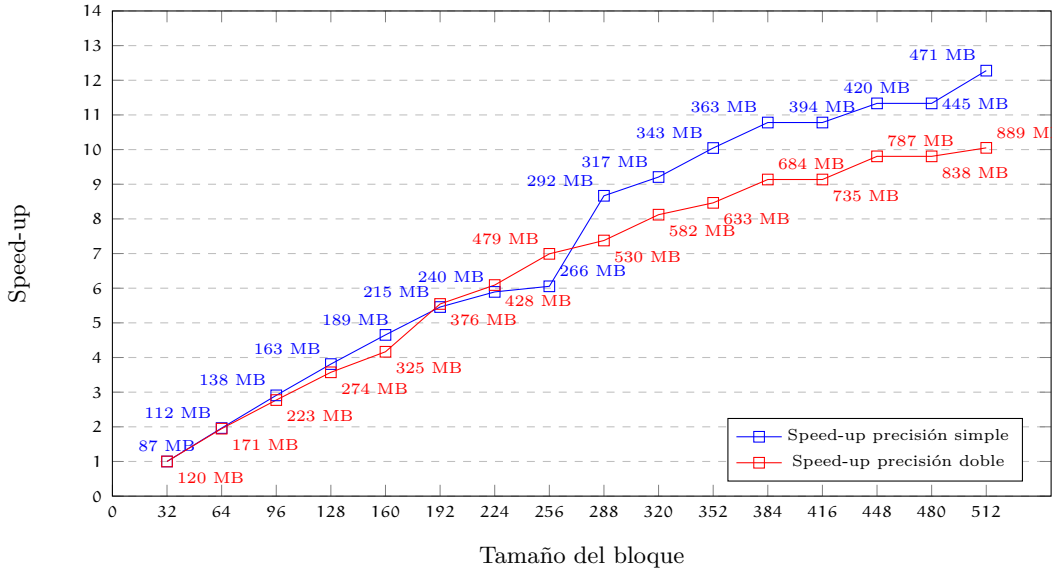


Figura 5.18: Comportamiento del speed-up incrementando el número de hilos por bloque en doble y precisión sencilla, con su respectiva cantidad de memoria requerida.

El número máximo de hilos con su propio espacio de memoria que pueden crearse es de 7,168 (14 bloques $\times$ 512 hilos), con un tamaño de $150 \times 100 = 15\,000$ elementos, que junto con el número de prismas que deben almacenarse previamente en la memoria de la tarjeta, requiere 889 MB en precisión doble y 471 MB en precisión

sencilla. Y se observa que la cantidad de memoria requerida en la GPU aumenta, pero mejora el rendimiento.

El número de bloques y su tamaño (el número de hilos creados) dependerá de la tarjeta que se utilice. Es importante mencionar que existen muchas tarjetas de rango medio que no superan los 500 MB de memoria, por lo que no se podrían crear muchas mallas y el rendimiento esperado se ve decrementado.

Examinando el rendimiento si fijamos el número de hilos en 32 por bloque y variamos el número de bloques de 14 a 224 en múltiplos de 14.

En la Tabla 2 se tabulan los resultados de los tiempos de ejecución cuando se crean bloques de 32 hilos en múltiplos de 14, donde se puede observar que se alcanza el mínimo en tiempo de ejecución, tanto para precisión sencilla como doble con 224 bloques. En la Tabla 2 se representan los datos mostrados en la Tabla 5.19 en la que podemos observar que al sobrepasar la creación de 112 bloques existe una baja en rendimiento para volver a mejorar el rendimiento. Esta baja en el rendimiento se puede observar claramente cuando se calcula el correspondiente speed-up mostrado en la Figura 5.20.

Comparando con la Figura 5.17, el comportamiento producido por el aumento del número de bloques no es tan estable como el aumento del número de hilos por bloque ya que la ocupación no aumenta debido a que sólo se maneja un warp (32 hilos) por bloque.

Tabla 2: Tiempos de cómputo obtenidos y memoria utilizada usando OpenMP y CUDA y una malla en múltiplos de 14 con precisión simple y doble.
PS-Precisión Simple, PD-Precisión Doble

Tamaño de la malla	Tiempo PS	Tiempo PD	Memoria PS	Memoria PD
14	442s	806s	87 MB	120 MB
28	222s	413s	112 MB	171 MB
42	147s	272s	138 MB	223 MB
56	111s	216s	163 MB	274 MB
70	89s	174s	189 MB	325 MB
84	75s	149s	215 MB	376 MB
98	66s	132s	240 MB	428 MB
112	57s	115s	266 MB	479 MB
126	98s	193s	292 MB	530 MB
140	89s	177s	317 MB	582 MB
154	81s	144s	343 MB	633 MB
168	75s	152s	368 MB	684 MB
182	72s	141s	394 MB	735 MB
196	66s	132s	420 MB	787 MB
210	60s	121s	445 MB	838 MB
224	57s	115s	471 MB	889 MB

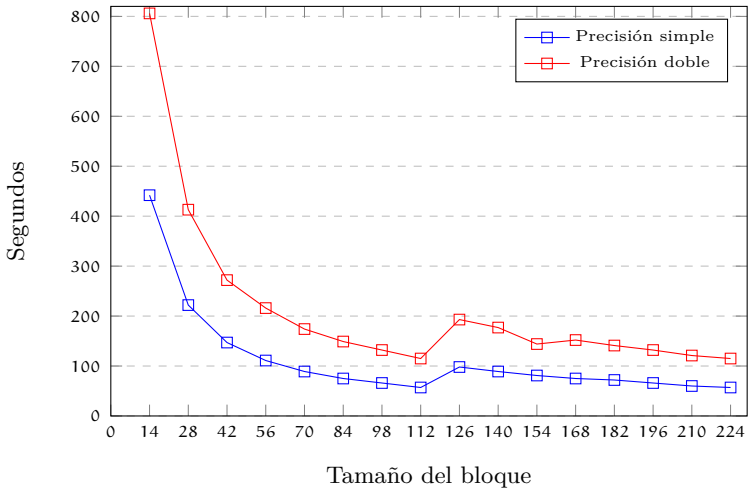


Figura 5.19: Tiempo de cómputo obtenido al variar el número de bloques.

Finalmente podemos concluir que al crear 32 bloques de 512 hilos es mejor que crear 224 bloques de 32 hilos, se muestra que esta última opción es más lenta porque aumenta el tiempo de ejecución en un 42 % para precisión doble y 54 % para precisión simple. Para este experimento la creación de varios bloques no es tan eficiente como aumentar el número de hilos por bloque.

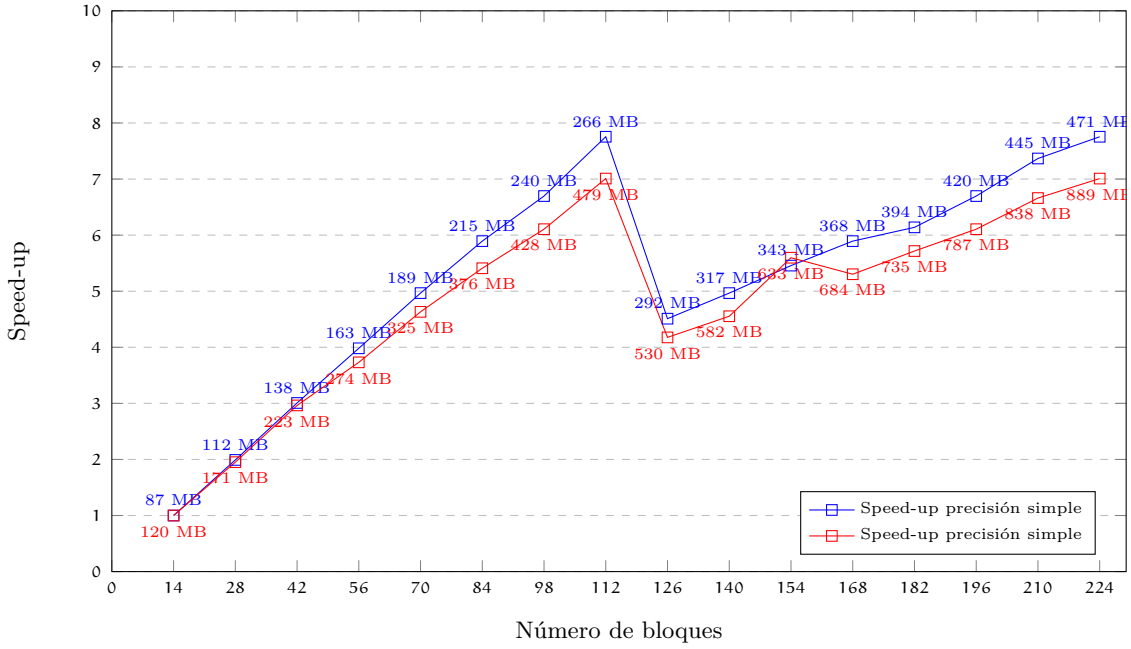


Figura 5.20: Speed-up.

5.3 RENDIMIENTO UTILIZANDO MÚLTIPLES GPUS

Para continuar con el experimento se analizó el rendimiento usando varias tarjetas (GPUs) en la misma workstation, las tarjetas se intercomunican a través del bus PCI-E y se intercomunican a través de OpenMP (memoria compartida). De acuerdo con los experimentos de la sección anterior la mejor configuración encontrada para este problema es crear 14 bloques con 512 hilos, por lo que ésta configuración utilizada.

Los resultados en tiempo del experimento, con respecto al tiempo utilizando tres GPUs integradas dentro de la misma workstation son presentados en la Tabla 3.

Tabla 3: Tiempos de cómputo obtenidos utilizando tres GPUs en precisión sencilla y doble. PS-Precisión Simple, PD-Precisión Doble

No. Tarjetas	Tiempo PS	Tiempo PD
1	36s	81s
2	16s	40s
3	12s	27s

En la Figura 5.21 se consideró a una tarjeta como una unidad de procesamiento y se puede observar que el tiempo se reduce casi proporcionalmente al número de tarjetas utilizadas y que en el caso particular de la precisión sencilla el tiempo se reduce prácticamente a la mitad.

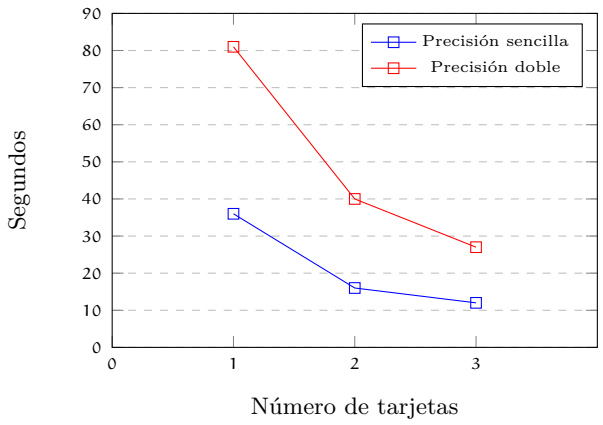


Figura 5.21: Comparación del tiempo de ejecución,utilizando tres tarjetas gráficas (C2070) en precisión sencilla y doble.

En la Tabla 4 se tabula el speed-up obtenido al utilizar 1,2 y 3 tarjetas; en la Figura 5.22 se muestra la gráfica correspondiente y claramente se ve un comportamiento prácticamente lineal, lo que implica la sobrecarga por el uso de OpenMP, es decir, la intercomunicación es prácticamente nula.

Tabla 4: Speed-Up obtenidos utilizando tres tarjetas gráficas con precisión simple y doble. PS-Precisión Simple, PD-Precisión Doble

No. Tarjetas	Speed-up PS	Speed-up PD
1	1	1
2	2.25	2.025
3	3.0	3.0

Los resultados arrojan que la latencia de la comunicación es despreciable, ya que no hay sobrecarga debido al uso de OpenMP dado que las funciones utilizadas solo se requieren al final del cálculo para realizar las reducciones.

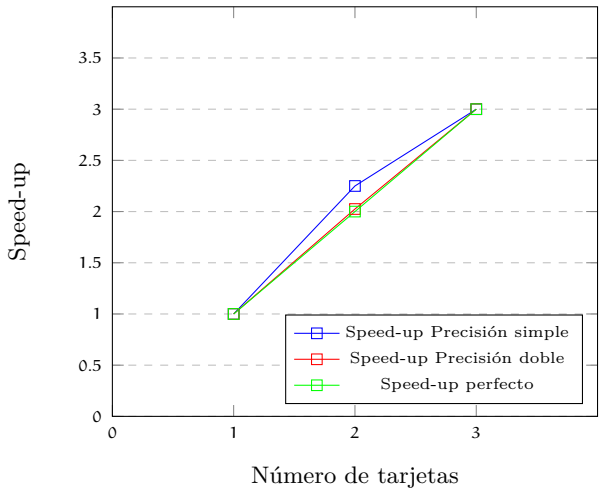


Figura 5.22: Speed-up obtenido con 3 tarjetas.

5.4 COMPARACIÓN CONTRA UN CLUSTER DE COMPUTADORAS

Para obtener una mejor perspectiva del rendimiento que se obtiene con el uso de tarjetas TESLA C2070 usando la implementación desarrollada CUDA + OpenMP, comparamos los resultados obtenidos con los de un programa previo diseñado para cluster en precisión doble.

El cluster es un equipo distribuido con las siguientes características:

- Nodo: 1 Procesador Intel Xeon modelo X5550 con cuatro procesadores físicos por procesador (2 hilos por núcleo),

- 29 nodos de procesamiento,
- Hyper-hilado activado,
- 40 GB de RAM por nodo,
- Sistema Operativo Linux RedHat

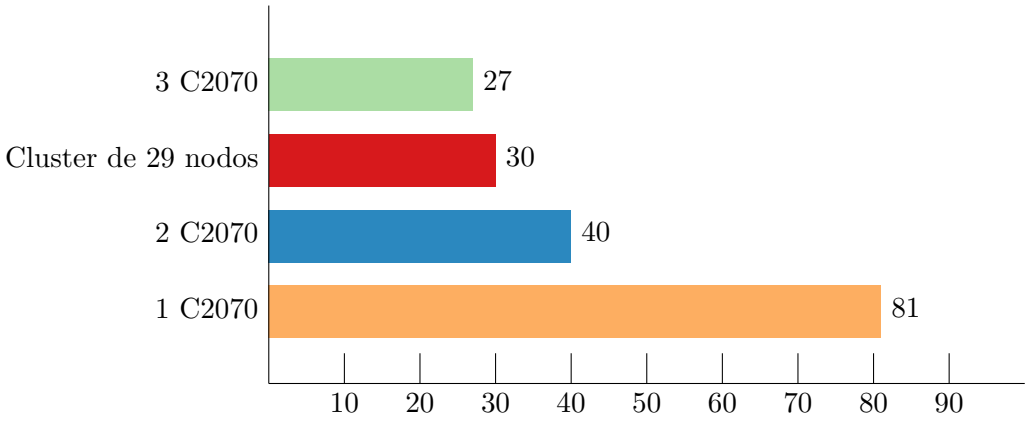


Figura 5.23: Tiempos de cómputo (segundos) obtenidos con 3 tarjetas y un cluster de 29 nodos.

Los resultados obtenidos con las tarjetas TESLA son comparados contra una versión bien optimizada MPI+OpenMP para el cálculo de la gravimetría de gravedad. La comparación de los resultados muestran que utilizando 29 nodos del cluster se requiere un tiempo de cómputo de 30 s para el problema de las esferas. En una tarjeta C2070 con la mejor configuración utilizada: 14 bloques y 512 hilos por bloque, el tiempo de ejecución es de 81 s, con dos tarjetas 40 s y con tres tarjetas requiere 27 s, por lo que se alcanza un rendimiento similar a un cluster de 29 nodos a un costo energético y de espacio muy bajo.

El problema que se ejecuta en el cluster es utilizando el caso de los de las esferas conformadas por 251,946 primas con 15,000 puntos de observación. El cluster es 2.7X más rápido que una tarjeta CUDA, pero si ocupamos las 3 tarjetas estas son 1.1X más rápidas que el cluster, considerando que las mediciones pueden variar entre 1 y 2 segundos, practicamente podemos decir que existe una equivalencia entre 3 tarjetas TESLA y el cluster de 29 nodos.

Considerando el consumo energético y el mantenimiento que implica tener un cluster de computadoras para esta aplicación la tecnología de GPUs es una excelente

alternativa por su bajo consumo energético, espacio y poco mantenimiento (Enos et al., 2010).

5.5 VALIDACIÓN DEL CÓDIGO NUMÉRICO

Los resultados producidos utilizando las tarjetas TESLA C2070 con respecto al rendimiento en tiempo son muy buenos, no obstante, es necesario verificar la calidad de la solución numérica, es decir, que los resultados numéricos obtenidos sean consistentes. Para tal efecto procedemos a comparar los resultados numéricos contra los resultados obtenidos en el cluster.

Para estimar la calidad de la solución se utilizó el error de norma L2 definido como:

$$e = \sqrt{\frac{1}{N_x N_y} \sum_{i=1}^{N_x} \sum_{i=1}^{N_y} | \, g_{i,j}^{gpu} - g_{i,j}^{cluster} \, |^2} \tag{34}$$

donde $g_{i,j}^{gpu}$ es el componente del tensor o el vector calculado en el GPU y $g_{i,j}^{cluster}$ es el correspondiente calculado en el cluster y N es el número de puntos en x y y de la malla de observación.

En la Tabla 5 se tabulan los errores del tensor del gradiente gravimétrico calculado en uno y tres GPUs, comparado contra el resultado del cluster.

Tabla 5: Errores de los componentes del vector gravimétrico, calculados en los GPUs en precisión doble precisión, con respecto a su contraparte en el cluster.

Componentes vectoriales	Error en un solo GPU - PD	Error con 3 Gpus - PD
g_x	1.7107e-09	1.7106e-09
g_y	1.1162e-09	1.1163e-09
g_z	2.0582e-09	2.0581e-09

En la Tabla 6 se muestran los errores de los componentes del tensor del gradiente gravimétrico calculados con uno y tres GPUs, comparado contra el resultado del cluster.

Tabla 6: Errores de los componentes del tensor de gravedad de gravedad en doble precisión, con respecto a su contraparte secuencial.

Componentes del tensor gravimétrico	Error en un solo GPU - PD	Error con 3 GPUs -PD
g_{xx}	7.7994e-11	7.8030e-11
g_{yy}	8.8905e-11	8.8935e-11
g_{zz}	4.6290e-11	4.6373e-11
g_{xy}	5.3476e-11	5.3503e-11
g_{xz}	9.2797e-10	9.2792e-10
g_{yz}	2.7225e-10	2.7207e-10

CONCLUSIONES

En Marzo del 2018 la compañía NVIDIA anunciaba su gama de computadoras DGX-1 y DGX-2, las cuales tienen capacidades de cómputo para ser consideradas como superordenadores personales centrados en tareas de cómputo numérico e inteligencia artificial. La máquina DGX-2 ofrece un poder de cómputo en precisión sencilla de 2 petaFLOPS en un entorno de dimensiones reducidas y bajo consumo energético, por medio de integrar 16 tarjetas Tesla V100 de arquitectura Volta con 32 GB de memoria interna, por un precio de $400 mil dólares americanos.

Con la aparición de la línea de máquinas DGX es notorio que la innovación y el constante desarrollo en el hardware de GPUs ha llevado a la creación de mini superordenadores que pueden ser instalados en una oficina, de hecho, los GPUs tienen actualmente tanta relevancia que se han convertido en pilar de los súper ordenadores para el desarrollo de aplicaciones en inteligencia artificial y cómputo científico. Su bajo costo energético y su reducido espacio los han convertido en una excelente herramienta computacional desde la aparición de CUDA C en el 2007.

De hecho el poder de los GPUs no está limitado solamente a equipos de gran escala, modelos como la Jetson AGX Xavier que en tan solo 10 cm^2 ofrecen el poder de 512 núcleos de procesamiento numérico con 16 GB de memoria, lo que permitiría darle a los drones o satélites nuevas capacidades de procesamiento.

Los recursos de cómputo que ofrecen los GPUs están orientados a satisfacer la demanda de aplicaciones numéricas intensivas en cómputo en las cuales se requiere procesar una gran cantidad de datos y se ha convertido en una opción para poder reducir los tiempos de cálculo, así como costos en la adquisición y mantenimiento de equipo científico especializado.

Debido a las prestaciones que ofrecen los GPUs, muchas aplicaciones se han migrado a CUDA, o hacen uso de GPUs para reducir los tiempos de cómputo, no solo en ingeniería y ciencia, incluso en el sector financiero donde software como; Numerix y CompatibL introdujeron soporte para CUDA y como resultado la velocidad de los cálculos ejecutados por dichas aplicaciones se incrementó cerca de 18 veces.

Con lo que respecta a la aplicación desarrollada en este trabajo, el diseño elegido muestra una reducción notable en el tiempo de ejecución para el cómputo del problema directo de la gravimetría, el problema de las esferas ejecutado en un solo núcleo de una computadora i7, consume alrededor de 2 horas para completarse, si comparamos contra los 27 segundos obtenidos con tres tarjetas C2070 obtenemos un factor sorprendente ya que muestra un rendimiento 266 veces más rápido, compara-

ble al factor obtenido contra un Cluster de 29 nodos pero a un consumo energético mucho más bajo, incluso como puede verse en los resultados obtenidos 3 tarjetas C2070 son aproximadamente equivalentes a un cluster de 29 nodos, por lo que el diseño e implementación se consideran satisfactorias.

Aún con el éxito obtenido en el rendimiento, es necesario señalar que la reducción en el tiempo de cómputo siempre va de la mano con un buen diseño y programación, aunado a un conocimiento profundo de la arquitectura, no conocer cómo funciona la arquitectura conlleva muchas veces a una baja de rendimiento, por este motivo es que para el programador inexperto puede ser frustrante no lograr la disminución del tiempo de cómputo. A pesar de ser arquitecturas poderosas para cómputo numérico, los GPUs como herramientas de cálculo requieren una alta comprensión de la arquitectura y la necesidad de programar a bajo nivel como con CUDA C, lo que puede convertirse en una tarea compleja y requiere un alto grado de conocimiento para obtener el mejor rendimiento posible, sin embargo, es necesario enfatizar que no todos los algoritmos se pueden migrar a GPU.

La clave en general para lograr un código eficiente para la GPU es el manejo correcto de los accesos a memoria llamada latencia que consiste en realizar el menor número posible de transferencias entre la memoria global del GPU y la memoria principal del CPU. Seguido por pocas llamadas a las funciones kernel CUDA. Lo que implica tener una gran cantidad de bloques para procesar una gran cantidad de hilos.

Con cada avance en la arquitectura de los GPU NVIDA aparece una nueva versión de CUDA C, la última versión al primer trimestre del 2018 es la 9, e incluye todos los avances con respecto a la arquitectura Volta, por lo que es necesario actualizarse constantemente para mantener los códigos compatibles con las nuevas versiones. También hay que considerar que las primeras arquitecturas van perdiendo soporte y los nuevos compiladores dejan de generar código para las tarjetas más antiguas.

Por lo expuesto, la programación en CUDA requiere un conocimiento profundo de la arquitectura para sacar en mayor provecho posible de la tarjeta. Aunque debe de tenerse en cuenta que análogamente a OpenMP que se ha convertido en un estándar para la programación basada en hilos, OpenACC se está convirtiendo en un estándar para la facilitar la programación en GPUs y se puede obtener gratuitamente a través de los compiladores de *PGI Community Edition*.

El diseño paralelo basado en multi-GPU sobre memoria compartida para el cálculo de los componentes vectoriales y tensoriales de la gravedad que fue desarrollado e implementado utilizando OpenMP y CUDA C por medio de una estructura bidimensional y una tridimensional para el manejo de la malla de observaciones y el ensamble de prismas respectivamente mostró un excelente rendimiento con base en los experimentos numéricos y las métricas obtenidas que validan la implementación reduciendo los tiempos de computo.

El Futuro de los GPUs parece promisorio y aunque en tecnología de cómputo es muy difícil predecir a futuro, por que las nuevas arquitecturas están influenciadas por distintos factores, como ciencia de materiales y el marketing, pero con mucha seguridad aparecerán plataformas basadas en la arquitectura de los GPUs, muy probablemente los procesadores futuros integrarán multiprocesadores gráficos con núcleos de procesamiento en la misma pastilla.

A

GLOSARIO Y ACRÓNIMOS

RISC Reduced Instruction Set Computer o Conjunto reducido de instrucciones de computadora. Un procesador RISC es aquel que tiene un conjunto de instrucciones como las aritméticas y lógicas.

SISD Single Instruction Single Data o Una instrucción Un Dato, un único procesador que ejecuta un sólo flujo de instrucciones.

MISD Multiple Instruction Single Data, Múltiples Instrucciones Un Solo Dato, muchas unidades funcionales realizan diferentes operaciones en los mismos datos.

SIMD Single Instruction Multiple Data o Una Instrucción Múltiples Datos, instrucciones que aplican una misma operación sobre un conjunto de datos.

MIMD Multiple Instruction, Multiple Data o Múltiples Instrucciones Múltiples Datos, instrucciones que aplican múltiples operaciones sobre un conjunto de datos.

API Application Programming Interface o Interfaz De Programación De Aplicaciones, es un subconjunto de rutinas, funciones y procedimientos que ofrece una biblioteca para ser utilizada por otro software.

TERAFLOP Es una medida utilizada para determinar cuántas operaciones de números en punto flotante puede hacer una computadora en un segundo.

CPU Central processing Unit o Unidad Central De Procesamiento se encarga de interpretar las instrucciones de un programa informático.

GPU Graphics Processing Unit o Unidad De Procesamiento Gráfico, es un coprocesador dedicado al procesamiento de operaciones de punto flotante.

ALU Arithmetic Logic Unit o Unidad Aritmético Lógica, calcula operaciones aritméticas (suma, resta, multiplicación, etc.) y operaciones lógicas.

HILO Es la secuencia más pequeña de instrucciones que puede administrarse de forma independiente por el programador.

HIPER-HILADO Permite que un solo procesador físico actúe como dos procesadores lógicos.

CLUSTER Conjunto de computadoras que se relacionan entre sí a través de una red de alta velocidad, actuando como una sola unidad.

ILP Instruction Level Parallelism o Paralelismo A Nivel De Instrucciones, es una técnica que consiste en ejecutar instrucciones de manera simultánea.

MULTI-NÚCLEO Un procesador que combina dos o más microprocesadores independientes en un solo circuito integrado.

SMT Simultaneous Multithreading o Multi-Hilado simultáneo, es una tecnología que permite simular dos procesadores lógicos dentro de un único procesador físico.

RAM Random Access Memory o Memoria de Acceso Aleatorio

WARP Es un grupo de 32 hilos que se ejecutan simultáneamente en un GPU.

MALLA Un conjunto de bloques de hilos.

KERNEL Una función o procedimiento ejecutado paralelamente en la GPU que es ejecutado por los hilos CUDA.

BIBLIOGRAFÍA

Al-Refaie, A. F., Yurchenko, S. N. and Tennyson, J. (2017). **G**PU **A**ccelerated **IN**tensities MPI (GAIN-MPI): A new method of computing Einstein-A coefficients, *Computer Physics Communications* **214**: 216 – 224.

Arroyo, M., Couder-Castañeda, C., Trujillo-Alcantara, A., Herrera-Diaz, I.-E. and Vera-Chavez, N. (2015). A performance study of a dual xeon-phi cluster for the forward modelling of gravitational fields, *Scientific Programming* **2015**. cited By 2.

Asenjo Plaza Rafael, Gutiérrez Carrasco Eladio, R. C. J. (2001). *Fundamentos de los computadores*, Universidad de Málaga.

Bruce Jacob, Spencer W., D. T. W. (2007). *Memory systems: cache, DRAM, disk.*, Morgan Kaufmann.

Calore, E., Gabbana, A., Kraus, J., Pellegrini, E., Schifano, S. and Tripiccione, R. (2016). Massively parallel lattice–Boltzmann codes on large GPU clusters, *Parallel Computing* **58**: 1 – 24.

Chai, L., Gao, Q. and Panda, D. K. (2007). Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system, *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, IEEE, pp. 471–478.

Chapman Barbara, Jost Gabriele, R. v. d. P. (2008). Using openmp, portable shared memory parallel programming.

Chen, T. and Zhang, G. (2018). Forward modeling of gravity anomalies based on cell mergence and parallel computing, *Computers and Geosciences* **120**: 1 – 9. **URL:** *http://www.sciencedirect.com/science/article/pii/S0098300417312700*

Couder-Castañeda, C., Ortiz-Alemán, C., Orozco-Del-Castillo, M. and Nava-Flores, M. (2013). Tesla gpus versus mpi with openmp for the forward modeling of gravity and gravity gradient of large prisms ensemble, *Journal of Applied Mathematics* **2013**.

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

Couder-Castaneda, C., Ortiz-Aleman, C., del Castillo, M. G. O. and Nava-Flores, M. (2013). Tesla gpus versus mpi with openmp for the forward modeling of

gravity and gradient of large prisms ensemble, *Hindawi Publishing Corporation Journal Applied Mathematics* (437357): 15.

Enos, J., Steffen, C., Fullop, J., Showerman, M., Shi, G., Esler, K., Kindratenko, V., Stone, J. E. and Phillips, J. C. (2010). Quantifying the impact of gpus on performance and energy efficiency in hpc clusters, *International Conference on Green Computing*, pp. 317–324.

Flynn, M. (1972). Computer organizations and their effectiveness, *IEEE TRANSACTIONS ON COMPUTERS* **C-21**: 948–960.

Foster, I. (1995). *Designing and building parallel programs*, Vol. 78, Addison Wesley Publishing Company Boston.

Galizia, A., D'Agostino, D. and Clematis, A. (2015). An mpi-cuda library for image processing on hpc architectures, *Journal of Computational and Applied Mathematics* **273**: 414–427.

Godse., A. (2009). *Computer Graphics*, Technical publications.

Harrigan., P. (2004). *First person: new media as story, performance and game.*, MIT Press.

Hernández-Gómez, J., Couder-Castañeda, C., Grageda-Arellano, J., Ortiz Alemán, J., Solís-Santomé, A. and Medina, I. (2017). Remote sensing of gravity: Feasibility of low orbit local gravimetry with nanosatellites. cited By 0.

Huang, C., Shi, B., He, N. and Chai, Z. (2015). Implementation of multi-gpu based lattice boltzmann method for flow through porous media, *Advances in Applied Mathematics and Mechanics* **7**(1): 1–12.

Jason Roberts, S. A. (2006). *Multi-Core Programming, Increasing Performance through Software Multi-threading*, Richard Bowles.

Kaczmarski, K., Przymus, P. and Rzazewski, P. (2015). Improving high-performance gpu graph traversal with compression, *Advances in Intelligent Systems and Computing* **312**: 201–214.

Kandort Edward, S. J. (2010). *CUDA By Example*, Addison-Wesley.

Krol, D., Harris, J. and Zydek, D. (2015). Hybrid gpu/cpu approach to multiphysics simulation, *Advances in Intelligent Systems and Computing* **1089**: 893–899.

Li Xiong, C. M. (1997). *Threedimensional gravity modeling in all space. SEG Technical Program Expanded Abstracts 1997: pp. 474-477*, Society of Exploration Geophysicists.

Lin, M., Xu, M. and Fu, X. (2017). GPU-accelerated computing for Lagrangian coherent structures of multi-body gravitational regimes, *Astrophysics and Space Science* **362**(4): 66.

Moorkamp, M., Jegen, M., Roberts, A. and Hobbs, R. (2010a). Massively parallel forward modeling of scalar and tensor gravimetry data, *Computers and Geosciences* **36**(5): 680 – 686.
URL: *http://www.sciencedirect.com/science/article/pii/S0098300410000579*

Moorkamp, M., Jegen, M., Roberts, A. and Hobbs, R. (2010b). Massively parallel forward modeling of scalar and tensor gravimetry data, *Geofisica Internacional* **36**: 680–686.

Nakata, N., Tsuji, T. and Matsuoka, T. (2011). Acceleration of computation speed for elastic wave simulation using a graphic processing unit, *Exploration Geophysics* **42**(1): 98–104.

Priimak, D. (2014). Finite difference numerical method for the superlattice boltzmann transport equation and case comparison of cpu(c) and gpu(cuda) implementations, *Journal of Computational Physics* **278**(1): 182–192.

Ramos, E. L. (2008). Geologiía general y de méxico.

Root., M. B. (1999). *DirectX Complete. Complete Series*, McGraw-Hill.

Strohm, P., Wittmer, S., Haberstroh, A. and Lauer, T. (2015). Gpu-accelerated quantification filters for analytical queries in multidimensional databases, *Advances in Intelligent Systems and Computing* **312**: 229–242.

Teodoro, G., Kurc, T., Kong, J., Cooper, L. and Saltz, J. (2014). Comparative performance analysis of intel (r) xeon phi (tm), gpu, and cpu: a case study from microscopy image analysis, *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, IEEE, pp. 1063–1072.

Von Neumann, J. P. E. and Mauchly, J. (1945). First draft of a report on the edvac., *W. Aspray and A. Burks* .

Wright., R. L. (2007). *OpenGL superbible: comprehensive tutorial and referente OpenGL Series.*, Edition 4. ed. Addison-Wesley.

Xu, C., Deng, X., Zhang, L., Fang, J., Wang, G., Jiang, Y., Cao, W., Che, Y., Wang, Y., Wang, Z., Liu, W. and Cheng, X. (2014). Collaborating cpu and gpu for large-scale high-order cfd simulations with complex grids on the tianhe-1a supercomputer, *Journal of Computational Physics* **278**(1): 275–297.

Zhang, T., Du, Y., Huang, T. and Li, X. (2014). Gpu-accelerated 3d reconstruction of porous media using multiple-point statistics, *Computational Geosciences* .