UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ACATLÁN

Estructuras de datos multi-dimensionales para cálculo numérico en CUDA C

TESIS

QUE PARA OBTENER EL TÍTULO DE:

LIC. EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN

PRESENTA:
JOSEFINA SÁNCHEZ NOGUEZ

ASESOR: DR. CARLOS COUDER CASTAÑEDA

SANTA CRUZ ACATLÁN, NAUCALPAN, EDO. DE MÉXIC, OCTUBRE 2021

Dedicatoria.

La culminación de una carrera no sería posible sin la colaboración de un conjunto de personas que nos guían y acompañan durante todo el proceso.

El esfuerzo realizado dentro de este trabajo de investigación va principalmente dedicado a mi asesor el Doctor Carlos Couder Castañeda, por el apoyo y herramientas brindadas.

De igual manera, a mis sínidos, profesores de la carrera y personal educativo que fueron los que me acompañaron en cada paso que di.

Por último pero no menos importante a mi esposo Rubicel por su apoyo incondicional en todo momento.

AGRADECIMIENTOS

A la Universidad Nacional Autónoma de México, Facultad de Estudios Superiores Acatlán y a la carrera de Matemáticas Aplicadas y Computación por la oportunidad que me dieron para mi formación.

A mi asesor el Doctor Carlos Couder Castañena por el apoyo y herramientas brindadas.

A los señores miembros del sínodo:

Mtra. Sara Camacho Cancino.

Mtra. Socorro Martínez José.

Mtra. Teresa Carrillo Ramírez.

Lic. Oscar Gabriel Caballero Martínez.

Por su tiempo y dedicación.

Al Centro de Desarrollo Aeroespacial del Instituto Politécnico Nacional, por el acceso a las tarjetas gráficas para poder llevar a cabo los experimentos requeridos en este trabajo.

RESUMEN

En años recientes, la necesidad de incrementar la velocidad de procesamiento de los algoritmos numéricos, ha conducido al uso de tarjetas gráficas conocidas como Unidades de Procesamiento Gráfico o GPUs por sus siglas en inglés (Graphic Procesing Units). En el 2007 NVIDIA a través de la plataforma CUDA (Compute Unified Device Architecture) permitió el acceso global a los GPUs con la finalidad de desarrollar aplicaciones de propósito general, que inicialmente estaban orientadas únicamente a tareas de visualización. CUDA es una de las interfaces de programación de aplicaciones más populares para acelerar una variedad de funciones en el GPU.

CUDA son las siglas en inglés Compute Unified Device Architecture (Arquitectura Unificada de Dispositivos de Cómputo) que hace referencia a una plataforma de computación en paralelo incluyendo un compilador y un conjunto de herramientas de desarrollo creadas por Nvidia qque permiten a los programadores usar una variación del lenguaje de programación C (CUDA C) para codificar algoritmos en GPU de Nvidia. (Ref. Wikipedia).

CUDA puede permitir que el código escrito en `C` o `C++` se ejecute de manera eficiente en una GPU con un esfuerzo de programación relativamente razonable. Logra un equilibrio entre la necesidad de conocer la arquitectura para explotarla bien, y la necesidad de tener una interfaz de programación que sea fácil de uso y resultados en programas legibles.

Las extensiones al lenguaje `C` que introduce CUDA facilita enormemente la programación de los GPUs, no obstante, la creación de arreglos multidimensionales no es tan simple como en lenguaje `C`, lo que implica que los códigos que utilizan arreglos bidimensionales o tridimensionales deben ser trabajados como unidimensionales requiriéndose un mapeo de los índices bidimensionales o tridimensionales a un índice unidimensional, lo cual puede ser propenso a errores, sobre todo cuando se implementan moléculas computaciones que representan diferencias finitas o elementos finitos, donde los índices juegan un papel crucial.

La implementación de arreglos multidimensionales en CUDA `C`, no es análogo a C estándar, por lo que es una pregunta recurrente en los foros de programación, por lo que en este trabajo presentamos una solución práctica para crear arreglos 2D y 3D en CUDA `C`, de la forma más parecida a estándar `C`, dando una posible solución para generar un código más legible a comparación de los que solo están basados en arreglos unidimensionales.

Para demostrar la capacidad de la estructura de datos propuesta, se resuelve la ecuación de transferencia de conducción de calor en una placa (2D) y en un cubo (3D), los resultados de rendimiento muestran que no existe una sobre carga significativa en el rendimiento y si una mejor lectura del código, además de que se prueba el acceso a memoria compartida. Para verificar los resultados se programó la contraparte serial en estándar C.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

La necesidad de tener computadoras cada vez más poderosas está ligada a los requerimientos de los problemas actuales de la computación científica, donde se necesita un gran poder de cómputo para modelar los fenómenos físicos, ingenieriles y de inteligencia artificial, la resolución numérica que demandan los modelos computacionales actuales requiere del uso de supercómputadoras, tal es el caso, de la simulación del clima, la propagación de ondas sísmicas, la formación de galaxias, la simulación de yacimientos, la dinámica molecular y diversas aplicaciones numéricas en ciencia e ingeniería que cada vez demandan un mayor poder de cómputo a bajo costo energético y mantenimiento. Para satisfacer esta demanda se introdujo las unidades de procesamiento gráfico de propósito general (GPUs), esto implicó un gran cambio en la arquitectura de los equipos y no solo eso, si no también del paradigma de programación.

Los GPUs claramente no son tecnología reciente, de hecho tienen sus orígenes en los 70s, y son responsables de la manipulación de los gráficos y la salida a pantalla, sin embargo, su poder realmente se ve en la década de los 90s cuando Sony introduce un GPU dedicado en su PlayStation, y en 1999 NVIDIA introduce su primer GPU, la GeForce 256 con poder de renderización y su competencia la ATI Radeon 9700 de AMD aparece en el 2002. No obstante, el cambio real de paradigma se introduce en el 2007 con el desarrollo de CUDA (Compute Unified Device Architecture), que hace referencia a la Arquitectura de Dispositivos Unificados (Nvidia, 2007), la cual permitió utilizar los GPUs para aplicaciones de propósito general y no solo para gráficos. Con esta apertura a la plataforma de los GPUs y con el desarrollo de extensiones al lenguaje C, llamado CUDA C, la programación sobre estos dispositivos creció enormemente y se empezaron a migrar códigos con la finalidad de acelerar los procesos de cómputo, finalmente la característica más importante que ofrece el GPU con respecto al CPU es la velocidad de procesamiento de aplicaciones numéricas.

Algunas aplicaciones que se han migrado a CUDA son: 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) entre otros.

Aunque los aceleradores gráficos pueden hacer que muchas aplicaciones aceleren su ejecución a un bajo costo energético, no dejan de existir inconvenientes:

1.- No todas las aplicaciones se pueden migrar, solo se pueden ver beneficiadas las que requieren un alto grado de paralelismo.

2.- El código se tiene que reescribir totalmente ya que esta fuertemente ligado a la arquitectura, lo que implica que la programabilidad es más compleja y finalmente que una tarjeta de alto rendimiento es relativamente costosa.

Actualmente la construcción de supercómputadoras que integran GPUs dentro de sus nodos es cada vez más común, debido principalmente a que existe una necesidad inherente de incluirlas para aumentar el rendimiento a un bajo consumo energético. En la lista de las 500 supercomputadoras más poderosas (www.top500.org), se encuentra como base de su poder los GPUs.

Finalmente puede concluirse que la introducción de los GPUs como dispositivios de propósito general, 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 multicomputadoras (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), aunado a que su desarrollo ya fue descontinuado.

Este trabajo esta orientado a la programabilidad de los GPUs, y propone una estructura de datos para dar respuesta a una pregunta recurrente en los foros de programación CUDA: ¿Cómo crear arreglos 2D en CUDA como en C estándar? (`https://forums.developer.nvidia.com/t/how-to-cudamalloc-two-dimensional-array/4042`).

Para tal fin, se desarrolla una estructura de datos y se resuelve la ecuación de difusión de calor en diferencias finitas con la finalidad de probarla, no solo en el aspecto de legibilidad del código, también en el rendimiento, por lo tanto, la hipótesis se basa en que es posible desarrollar una estructura de datos para permitir manejar de manera nativa arreglos multidimensionales en CUDA C (2D y 3D), con una

sintaxis similar a la del lenguaje C tradicional, mediante el redireccionamiento de direcciones de memoria dentro del GPU, mejorando considerablemente la legibilidad y sin introducir latencias que impacten en el rendimiento.

Se debe tener en cuenta que la incorporación de los GPUs, implicó un cambio en el paradigma de programación, y en consecuencia en el diseño de aplicaciones numéricas de propósito científico e ingenieril, estas tuvieron 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; cada versión nueva de CUDA exige una mayor versión del hardware, dejando obsoletas a algunas arquitecturas previas.

Actualmente CUDA se encuentra en la versión 11.2 y la arquitectura de hardware más avanzada 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.

El vertiginoso crecimiento que ha tenido la arquitectura de los GPUs, ha ocasionado que constantemente las interfaces de programación se actualicen, 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.

Por lo tanto, el objetivo general de este trabajo consiste en implementar una estructura de datos en CUDA `C` que permita el manejo de arreglos multi-dimensionales con una sintaxis similar al lenguaje `C` estándar, con la finalidad de mejorar el desarrollo y mantenimiento de aplicaciones escritas a bajo nivel para cálculo numérico y probar la fiabilidad y rendimiento de dicha estructura.

El trabajo esta organizado como sigue:

Capítulo 1.- Se aborda el estado del arte de todo lo relacionado con la arquitectura paralelas actuales, con la finalidad de contextualizar los GPUs NVIDIA utilizados en este trabajo.

Capítulo 2.- Se analiza la plataforma y el paradigma de programación de la plataforma CUDA basada en tarjetas gráficas NVIDIA.

Capítulo 3.- Se establece la estructura de datos propuesta mediante técnicas de transferencia de direcciones de memoria, que permitan manejar arreglos multidimensionales de manera nativa y con sintaxis similar a la del lenguaje C tradicional.

Capítulo 4.- Se dan las pruebas de rendimiento de las estructuras desarrolladas utilizando como aplicación la ecuación de transferencia de calor resuelta por dife-

rencias finitas, enseguida, se exponen los comentarios con respecto a los resultados obtenidos en tiempo de cómputo y las ventajas con las estructuras desarrolladas.

1

ARQUITECTURAS PARALELAS

En este capítulo se introduce a las arquitecturas paralelas actuales con la finalidad de dar contexto al uso de los GPUs.

1.1 ANTECEDENTES

La computación paralela se puede considerar más que una estrategia para alcanzar un alto rendimiento para reducir los tiempos de cómputo, de hecho, es un nuevo paradigma de programación en la cual se puede escalar desde dos procesadores hasta el poder de cómputo ofrecido por las máquinas más poderosas construidas para este propósito, en en sitio web (www.top500.org) pueden ser consultadas las 500 computadoras más poderosas que se van construyendo y se actualiza constantemente.

En la década de los 80's el incremento de las capacidades de una computadora estaba basado en incrementar la velocidad del ciclo de reloj y mejorar el ancho de banda de la transferencia de datos. No obstante, esta idea fue cambiada con la introducción del procesamiento en paralelo, que básicamente consiste en dos o más unidades de procesamiento trabajando en conjunto con la finalidad de resolver un problema dado en el menor tiempo posible. Hasta hace unas décadas los procesadores escalaban frecuentemente su velocidad de frecuencia, los procesadores de apenas unos cientos de MHz evolucionaron hasta superar 1Ghz de velocidad en unos cuantos años.

El primer procesador de Intel que superó la barrera de 1 GHz fue el Pentium 4 de 1,3 GHz lanzado en 2001, y posteriormente se escaló hasta los 2,4 GHz. Con la segunda generación de procesadores el Core i7-2600 alcanzaba los 3,4 GHz de base y 3,8 GHz en modo Boost, no obstante, en la última década, la frecuencia no ha seguido creciendo al mismo ritmo (Ross, 2008).

Con la finalidad de tener un mayor poder de cómputo, la solución fue la introducción del procesamiento en paralelo, que consiste en unir dos más computadoras para resolver un problema computacional y básicamente existen 3 tipos de arquitecturas: de memoria compartida, memoria distribuida y las hibridaciones entre estas dos (Blazewicz et al., 2012).

En los últimos cincuenta años el campo de la computación científica ha motivado el uso y el desarrollo de la computación paralela, produciéndose arquitecturas y

sistemas de software específicos para esta área, como las supercomputadoras de procesamiento simétrico (SMP) y las multicomputadoras conectadas por una red externa de alta velocidad (cluster).

La computación paralela ha transformado varias disciplinas de la ciencia y de la ingeniería, incluyendo la oceanografía, la modelación ambiental, la mecánica molecular, la sismología y la turbulencia. Por ejemplo, en la cosmología ha permitido el estudio de la evolución y estructura del universo, simulando los procesos físicos que en teoría dieron origen a las estrellas y planetas, con las simulaciones es posible descartar ciertas teorías de la formación del universo comparándolas con la gran cantidad de observaciones captadas por los telescopios Hubble y Digital Sky Suver (Evrard et al., 2002).

Diseñar un código para una computadora paralela es más que simplemente reescribir una versión existente para una nueva máquina, debido a que las máquinas paralelas son fundamentalmente diferentes de sus predecesores secuenciales y vectoriales, adicionalmente rediseñar el código proporciona la oportunidad de reformular el código básico, las estructuras de datos y lo más importante, revisar la representación de los procesos o la dinámica involucrados. Por ejemplo, en la modelación oceánica, el código de Bryan-Cox-Semtner (BCS) fue reestructurado para que se pudiera ejecutar en una arquitectura vectorial como las computadoras Cray CM-2 y CM-5. Sin embargo, éste fue ineficiente en paralelo por dos razones principales: la estructura de los ciclos necesitaba ser reorganizada y la lógica de las comunicaciones requerían una nueva formulación (Akl, 2000). Análogamente la generación de código para los GPUs es similar, ya que prácticamente nada del código previamente escrito puede ser reutilizado.

1.2 TAXONOMÍA DE FLYNN

Existen diferentes tipos de computadoras paralelas, y los modelos de los fabricantes han venido y pasado de moda a través de los años. Esto es un proceso de evolución y mercadotecnia en la tecnología del hardware, que ha permitido integrar procesadores más rápidos y pequeños en las computadoras. Estos procesadores son cada vez más económicos debido a que son producidos en masa y por consecuencia se abarata el costo de los sistemas multiprocesador. En una computadora paralela el sistema de interconexión entre procesadores es medular en el sistema para reducir los tiempos de cómputo, y resulta ser relativamente más costoso que los nodos de procesamiento o procesadores.

Una computadora paralela es un conjunto de procesadores o nodos de procesamiento que trabajan sincronizadamente para resolver un problema computacional en común. Para que los procesadores puedan trabajar cooperativamente es necesario

que estén conectados por medio de una red. Si la red se forma por conexiones dentro de una computadora, se le conoce como sistema de procesamiento simétrico SMP; por otro lado, si la red se forma por conexiones entre diferentes computadoras, a dicho sistema se le llama cluster o multicomputadora (Baker and Buyya, 1999).

El tipo de sistema paralelo determina la configuración de la memoria, si se trata de un sistema SMP, la memoria es compartida y los procesadores tienen acceso a cualquier localidad de memoria, siendo las direcciones únicas e idénticas.

Por otra parte, un sistema multiprocesador maneja memoria distribuida, en la que cada procesador o nodo tiene su propia memoria privada, comunicándose por medio de una red de interconexión basada en alguna topología, generalmente de alta velocidad para reducir la latencia de comunicación.

Se puede clasificar a los sistemas computacionales por medio del flujo de instrucciones y de datos que procesan, de acuerdo a la taxonomía de Flynn de la siguiente forma (Snyder, 1988):

- SISD. (Single Instruction Single Data) una instrucción, un solo dato.
- MISD. (Multiple Instructions Single Data) múltiples instrucciones, un solo dato (arquitectura no existente).
- SIMD. (Single Instruction Multiple Data) una instrucción, múltiples datos.
- MIMD. (Multiple Instructions Multiple Data) múltiples instrucciones, múltiples datos.

El tipo SISD refiere a las computadoras seriales y es la arquitectura tradicional de von Neumann, en la que un procesador ejecuta un solo flujo de instrucciones y cada instrucción opera sobre un solo dato. La siguiente arquitectura SIMD consiste en aplicar una instrucción a un conjunto de datos en sucesión, por ejemplo, sumar el número 2 a todos los elementos de un arreglo. El tipo SIMD puede ser subdividido en procesadores vectoriales y arreglo de procesadores, en ambos tipos, una sola instrucción resulta en operaciones idénticas llevadas a cabo sobre diferentes datos. En máquinas vectoriales antigüas como la Fujitsu VPP300 y la CRAY-YMP, un solo procesador funciona como una computadora SIMD, en contraste, un arreglo de procesadores consiste en unidades simples de procesamiento controladas por un procesador maestro. El arreglo de procesadores para procesamiento SIMD actualmente ya es una arquitectura obsoleta, pero se sigue manteniendo como un conjunto de instrucciones con las tecnologías MMX y SSE en sus últimas versiones, que dan soporte a procesamiento vectorial sobre enteros y flotantes respectivamente.

La arquitectura SIMD presenta un comportamiento distinto a la MIMD al incrementar el número de operaciones independientes que se demanden, para ejemplificar, supóngase, que se tiene una matriz, a la que se quiere multiplicar por -1, y después sumar 3 a cada uno de sus elementos. En una arquitectura SIMD, estás operaciones

deben hacerse en dos pasos. En el primero el procesador de control indica que se deben multiplicar por -1 a cada uno de los elementos de la matriz; y en el segundo indica sumar 3 a los elementos resultantes. Esto es un inconveniente, ya que mientras se multiplica no se puede sumar, por lo que no se puede avanzar solo hasta que una operación sea realizada sobre todos los elementos. Al tipo de arquitectura SIMD pertenecen los GPUs.

La solución a la limitante que presenta la arquitectura SIMD consiste en construir arquitecturas multi-núcleo, un cluster multiprocesador, en el que cada procesador ejecuta su propio flujo de instrucciones y trabaja sobre sus propios datos. La arquitectura MIMD en combinación con la SIMD es la utilizada en las modernas de supercomputadoras, debido a su flexibilidad y que a los fabricantes les favorece económicamente construir tales arquitecturas, combinando varios microprocesadores con capacidades SIMD. Entre más heterogénea una arquitectura es más complicado construir un programa, que para una SISD o SIMD.

En las arquitecturas MIMD podemos encontrar las máquinas de procesamiento simétrico (SMP) y los clusters. En las arquitecturas SMP los procesadores están fuertemente aclopados y controlados por una sola instancia del sistema operativo como es el caso de los sistemas SMP basados en Intel Xeon MP. En los clusters los procesadores no necesariamente son del mismo tipo, de hecho pueden ser de diferente arquitectura y velocidad, aunque es inusual que así suceda.

1.3 ARQUITECTURAS DE MEMORIA COMPARTIDA

Alrededor del año 2005, la arquitectura de las computadoras hicieron uno de sus mayores avances al incluir varias copias del mismo procesador dentro del mismo chip, estas copias son llamados núcleos; un procesador multi-núcleo (multi-core) es un circuito integrado con dos o más procesadores, con la finalidad de aumentar el rendimiento y reducir el consumo energético, permitiendo la ejecución eficiente, de múltiples tareas simultaneas, por ejemplo, una configuración doble-núcleo (dual-core) es comparable a tener dos procesadores de manera independiente, pero instalados en la misma computadora, y en el mismo socket por lo cual la comunicación entre ellos es extremadamente rápida, de manera teórica un procesador dual-core tiene el doble de rendimiento que un procesador single-core, aunque en la pŕactica es improbable que se cumpla.

Para superar el número de cores incluidos se crearon las arquitecturas de procesamiento simétrico, las cuales incluyen 2, 4, 8, 16 y 32 procesadores idénticos que comparten una memoria principal, los procesadores son chips separados con múltiples núcleos en su interior, una de las máquinas más grandes construidas con esta arquitectura tiene 896 núcleos, basado en 32 procesadores Xeon Platinum 8180 con

28 núcleos cada uno. Estos sistemas permiten ejecutar más hilos de ejecución simultáneamente que con un solo procesador multi-núcleo, desafortunadamente esta clase de sistemas entre mas sockets tengan, se incrementa su costo de producción, por lo cual no son tan comunes, ya que principalmente la memoria principal cumple con más requisitos que una memoria RAM convencional, sin embargo, la programabilidad es relativamente más sencilla que los sistemas de memoria distribuida.

1.4 ARQUITECTURAS DE MEMORIA DISTRIBUIDA

Una representación clara de las arquitecturas de memoria compartida son los clusters de computadoras, los cuales son básicamente dos o más computadoras conectadas por una red, generalmente de alta velocidad con la finalidad de resolver un problema en común. Sin embargo, esta idea no debe confundirse, con el modelo cliente-servidor, en un cluster el objetivo es juntar el poder de varios nodos de cómputo, para combinar el poder computacional, en general los nodos son del mismo tipo, los cuales pueden ser de arquitectura de memoria compartida, es decir, máquinas SMP, y además están conectados por una red de alta velocidad. Este tipo de configuraciones son las más utilizadas en la computación de alto rendimiento, no obstante, existen diferentes configuraciones de menor costo, como clusters, basados en arquitectura ARM, o con computadoras tipo desktop. Los clusters para HPC, requieren administración, control de los sistemas de poder, sistemas de respaldo, redes basadas en fibra óptica, lo que aumenta su costo de mantenimiento, por lo cual no se puede subestimar la importancia de la gestión de clusters. En general, un programa para HPC normalmente esta enfocado a la reproducción de modelos numéricos o la investigación de información de instrumentación lógica, debido a su gran poder cómputo permiten analizar una gran cantidad de variables y llevar a cabo simulaciones complejas.

1.5 GRANULARIDAD DEL CÓDIGO Y NIVELES DE PARALELISMO

En las computadoras actuales el paralelismo aparece en varios niveles en el hardware y software. Por ejemplo, a nivel de circuitos, las señales viajan en paralelo a través de los buses de datos, en un nivel un poco más alto, las unidades de operación trabajan de manera paralela para mejorar el rendimiento, popularmente se conoce como *paralelismo a nivel de instrucciones*, este es el caso de los procesadores multi-núcleo, los cuales llegan a tener la capacidad de ejecutar varios procesos simultáneamente. Muchas computadoras utilizan técnicas de traslape para acceder a los recursos de entrada/salida y bancos de memoria que pueden ser accesados en paralelo para reducir el tiempo de acceso.

En el siguiente nivel se tienen los sistemas SMP que tienen varios procesadores con varios núcleos que trabajan en paralelo y finalmente en el nivel más alto del paralelismo podemos encontrar equipos interconectados por una red externa que trabajan en cooperación, conocidos como clusters.

Los primeros dos niveles (señales y nivel de circuitos) del paralelismo son llevados a cabo a nivel del hardware por lo que se conoce como *paralelismo en hardware*. Los siguientes dos niveles (componentes y sistemas) del paralelismo pueden ser expresados de manera implícita y/o explícita utilizando varias técnicas de programación. Los niveles de paralelismo también pueden estar presentes en la forma de estructurar el código fuente, estos niveles determinan la granularidad de la aplicación paralela. La Tabla 1.1 muestra las categorías de granularidad del código que se pueden paralelizar.

Tabla 1.1: Niveles de granularidad.

Granularidad	Nivel de código	Paralelizado por
Muy fina	Instrucción	Procesador
Fina	Estructura de control	Compilador
Media	Función o Rutina	Programador
Gruesa	Proceso o hilo de ejecución	Programador

Todas las aproximaciones para llevar a cabo el paralelismo en distinta granularidad del código tienen un objetivo común, aumentar la eficiencia del procesador reduciendo la latencia y el tiempo que consume accesar a la memoria central y a los dispositivos de almacenamiento. Para reducir la latencia el procesador puede realizar otra actividad mientras una operación de entrada ocurre para aprovechar al máximo el tiempo del procesador.

El paralelismo en una aplicación puede ser detectado en diferentes niveles, que se listan a continuación:

- de granularidad muy fina (multiples instrucciones)
- de granularidad fina (a nivel de datos),
- de granularidad media (a nivel de control),
- de granularidad gruesa (a nivel de tareas)

Los diferentes niveles de paralelismo se muestran en la Figura 1.1. Los primeros dos niveles son soportados transparentemente por el hardware y los compiladores que llevan a cabo la auto-paralelización. El programador en su mayor parte maneja los últimos dos niveles del paralelismo.

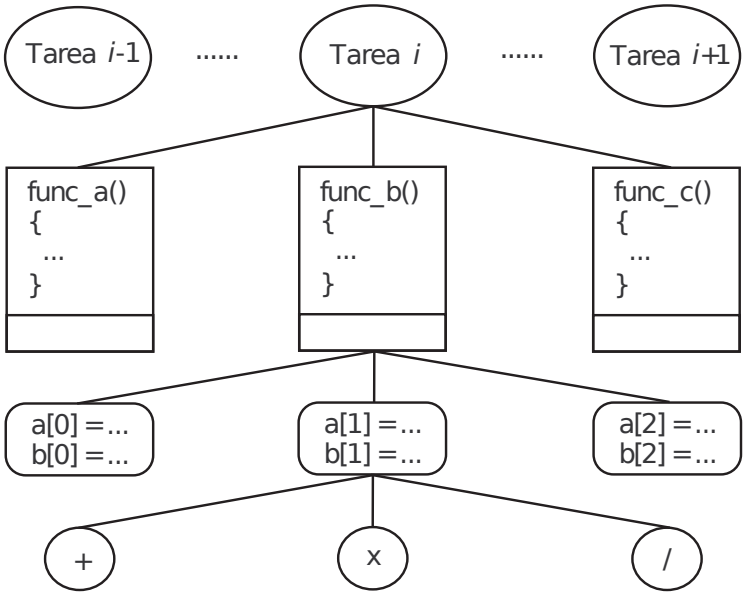


Figura 1.1: Niveles de paralelismo.

1.5.1 *Arquitectura de los GPUs*

Para entender la arquitectura de los GPUs, es necesario revisar cuales son las diferencias entre una unidad central de procesamiento (CPU) y una GPU. Una CPU común está optimizada para que sea lo más rápida posible para terminar una tarea con una latencia lo más baja posible, mientras mantiene la capacidad de cambiar rápidamente entre operaciones. Su naturaleza se trata de procesar tareas de manera serializada. Una GPU tiene que ver con la optimización del rendimiento, lo que permite impulsar tantas tareas como sea posible a través de sus componentes internos a la vez, por medio de procesar una tarea en paralelo. En la Figura 1.2 se muestra la diferencia entre el recuento de núcleos de una CPU y GPU, se enfatiza que el principal contraste entre ambos es que una GPU tiene muchos más núcleos para procesar una tarea.

Sin embargo, no se trata solo de la cantidad de núcleos, de hecho un CPU de 20 núcleos puede tener un rendimiento similar a un GPU de cientos. La gran cantidad de núcleos contenidos en el GPU es parte de su diseño y no son análogos a los núcleos del CPU, y cuando nos referimos a los núcleos en una GPU NVIDIA, nos referimos a núcleos CUDA que consisten en una especie de ALU (Unidad Aritmética Lógica) que opera en paralelo.

No obstante, a pesar de que no existe una analogía entre los núcleos CPU y GPU, entre la arquitectura general de una CPU y GPU, podemos ver muchas similitudes

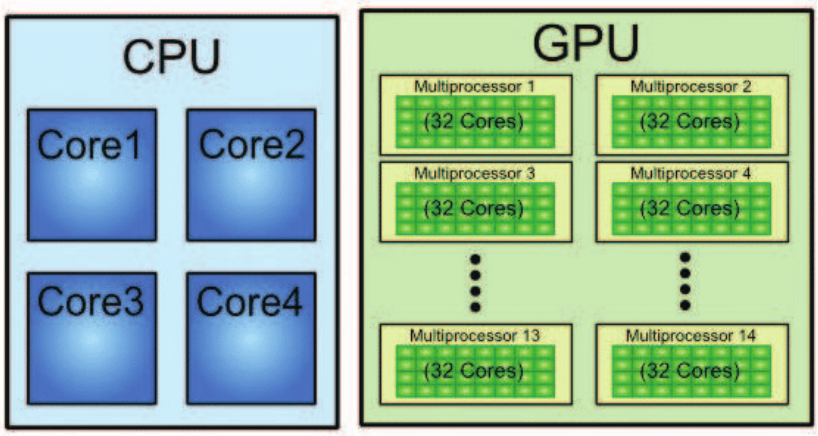


Figura 1.2: Número de núcleos, CPU vs GPU. Los núcleos en el GPU están organizados en multiprocesadores.

entre las dos. Ambos utilizan las construcciones de memoria de las capas de caché, el controlador de memoria y la memoria global. Una descripción general de alto nivel de las arquitecturas de CPU modernas indica que se trata de acceso a la memoria de baja latencia mediante el uso de importantes capas de memoria caché.

Un solo chip de CPU consta de núcleos que contienen cachés de capa 1, de instrucciones y datos separados, compatibles con la caché de capa 2. La caché de capa 3, o caché de último nivel, se comparte entre varios núcleos. Si los datos no residen en las capas de la caché, los obtendrá de la memoria DDR-4 global. La cantidad de núcleos por CPU puede llegar hasta 28 o 32 que corren hasta 2.5 GHz o 3.8 GHz con modo Turbo, dependiendo de la marca y modelo, los tamaños de caché varían hasta 2MB de caché L2 por núcleo.

Por su parte, la descripción general de la arquitectura de alto nivel de una GPU consiste en poner los núcleos disponibles a trabajar y se centra menos en el acceso a la memoria caché de baja latencia.

Un solo dispositivo GPU consta de cúmulos de procesadores que contienen varios multiprocesadores de transmisión (SM). Cada SM aloja una capa de caché de instrucciones de capa 1 con sus núcleos asociados. Por lo general, un SM utiliza una caché de capa 1 dedicada y una caché de capa 2 compartida antes de extraer datos de la memoria GDDR-5 global. Su arquitectura es tolerante con la latencia de la memoria (Owens, 2007).

En comparación con una CPU, una GPU funciona con menos capas de caché de memoria y son relativamente pequeñas. La razón es que una GPU tiene más transistores dedicados a la computación, lo que significa que le importa menos cuánto tiempo se tarda en recuperar los datos de la memoria. La latencia de acceso a la

memoria potencial está enmascarada siempre que la GPU tenga suficientes cálculos en ejecución, manteniéndola ocupada.

1.6 PROGRAMACIÓN PARALELA

El principal reto de la programación paralela es descomponer el programa en componentes que pueden ser ejecutadas en paralelo, no obstante, el nivel de descomposición va estrechamente influenciado con el tipo de arquitectura de la máquina paralela. Principalmente, se tienen dos tipos de estrategias, la descomposición por tareas (descomposición funcional) y la descomposición por datos (descomposición del dominio). La descomposición del dominio es la estrategia más utilizada para desarrollar programas científicos en máquinas paralelas, en este caso, la aplicación se descompone por dividir los datos sobre los cuales se opera entre las unidades de procesamiento, como es el caso de los GPUs. Típicamente esta estrategia involucra el compartir datos en las fronteras de los dominios que cada procesador maneja y el programador es el responsable de asegurar que estos datos sean compartidos y manejados correctamente, esto es, que los datos manejados en un procesador y utilizados por otro estén correctamente sincronizados.

1.6.1 *Descomposición de programas*

La implementación de un programa para una computadora paralela, implica lidiar con cuatro principales problemas.

1. Identificar los componentes que se pueden ejecutar de forma segura en paralelo.
2. Adoptar una estrategia para descomponer el programa.
3. Seleccionar el modelo de programación, lenguaje y en su caso la librería paralela.
4. Elegir el estilo de implementación que debe trabajar bien para el tipo de modelo de programación seleccionado.

Para la gran mayoría de los problemas que se abordan con programación paralela, existen diferentes estrategias para resolverlos, y la estrategia de paralelización que se seleccione debe ser mejor, al menos, que su contraparte secuencial. La metodología del diseño de un programa paralelo debe pretender alejarse de los aspectos de la arquitectura de la máquina, y aún cuando no es del todo posible, por la gran variedad de arquitecturas, algunas mejores que otras para determinados problemas, debe seguirse un proceso de diseño para su creación como todo software, en un principio lo más alejado posible de la arquitectura.

La metodología de diseño de Foster (Computing and Foster, 1995), llamada PCAM (ver Figura 1.3), por la primera letra de sus etapas, propone que la construcción de un programa paralelo debe contemplar las siguientes fases:

- Particionamiento.
- Comunicación.
- Aglomeración.
- Mapeo.

Las dos primeras se enfocan sobre el paralelismo y la escalabilidad, y por consecuencia en la búsqueda de algoritmos para cubrir estas cualidades. En las dos últimas, la atención es centrada sobre la localidad y las cuestiones relacionadas con el rendimiento.

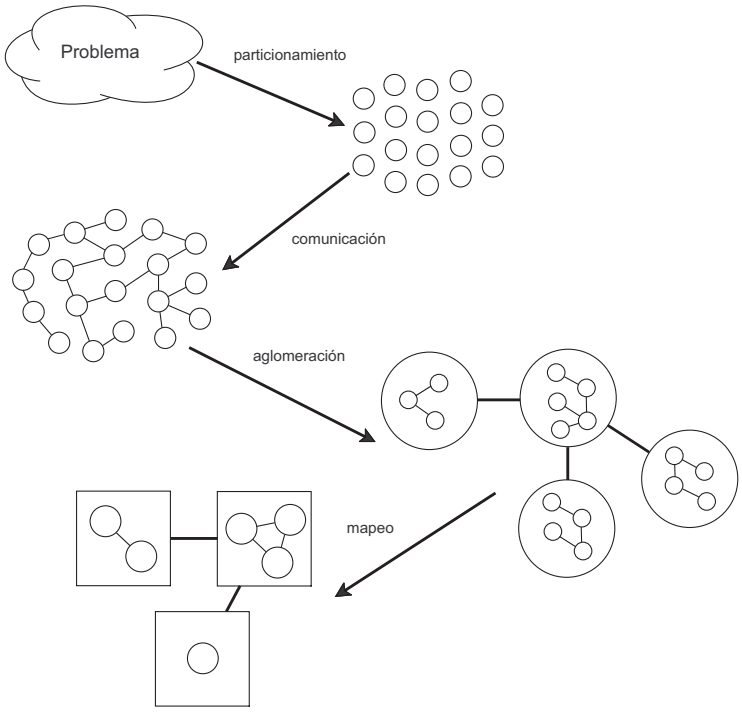


Figura 1.3: Metodología PCAM (Foster, 2020).

El resultado de aplicar la metodología PCAM, puede ser un programa que crea y destruye tareas dinámicamente, auxiliado por técnicas de balance de carga, con la finalidad de controlar el mapeo de las tareas a los procesadores, o puede ser un programa que mapea una tarea por procesador, aunque en algunos casos el mapeo puede ser predeterminado en la etapa de aglomeración.

En la etapa de particionamiento se definen la descomposición de las actividades de cómputo y de los datos sobre los cuales cada una de las tareas opera. La descomposición de los datos asociados a un problema se conoce como descomposición de dominio, y la descomposición en tareas se conoce como descomposición funcional.

La etapa de comunicación se enfoca al flujo de información y la coordinación entre tareas que son creadas en la fase de particionamiento. La naturaleza del problema y el método de descomposición determina los patrones de comunicación entre las tareas que conforman la ejecución en paralelo. Los patrones más populares de comunicación son: local/global, estructurado/no estructurado, estático/dinámico y sincronizado/no sincronizado.

En la etapa de aglomeración, las tareas y la estructura de la comunicación definidas en las dos primeras etapas son evaluadas en términos de los requerimientos de rendimiento y costos de implementación. Si es necesario, las tareas deben ser agrupadas en tareas de granularidad más grande para mejorar el rendimiento o reducir los costos de desarrollo, adicionalmente las comunicaciones individuales pueden ser agrupadas en supercomunicaciones, esto ayuda a reducir los costos de comunicación incrementando la granularidad de las comunicaciones, lo que se traduce en ganancia de flexibilidad en términos de escalabilidad y decisiones de mapeo, reduciendo los costos del diseño del programa.

Finalmente en la etapa de mapeo se asigna una tarea o varias a un procesador procurando maximizar los recursos computacionales de la máquina. Las decisiones de mapeo pueden ser estáticas, es decir, en tiempo de compilación, o dinámicas en tiempo de ejecución por medio de métodos de balance de carga.

1.6.2 *Modelos de programación paralela*

Existen tres herramientas principales para construir programas paralelos, a los que la mayoría se asemeja: `OpenMP`, `MPI`, `CUDA`. La elección de la herramienta generalmente depende de las siguientes características:

- Portabilidad.
- Facilidad de uso.
- Eficiencia.
- Costo.
- Conocimiento del programador.

El paradigma o modelo de programación paralela está fuertemente influenciado por el tipo de arquitectura paralela a la que se pretenda migrar, no obstante, se debe pretender que tenga cierto grado de programabilidad y transportabilidad, es decir, no

es factible escribir un programa diferente por cada tipo de arquitectura cada vez que esta cambie, y en cierta medida es verdadero, si la arquitectura no cambia demasiado, si tenemos un programa, que se ejecuta en un entorno de memoria compartida, se espera, es que el programa pueda transportarse a una arquitectura similar donde se aumente el número de procesadores sin tantos problemas de transportabilidad, pero si ésta cambia de memoria compartida a distribuida, se tiene que cambiar el modelo, porque la arquitectura es completamente diferente, más aún si se intenta migrar un programa de una arquitectura convencional o una arquitectura basada en GPU's.

1.6.2.1 *OpenMP*

`OpenMP` (`www.openmp.org`) es un conjunto de directivas y funciones contenidas en librerías disponibles para `FORTRAN`, `C` y `C++` , que permite desarrollar programas paralelos sobre máquinas de memoria compartida multiprocesador-multinúcleo. El desarrollo de aplicaciones paralelas en el ámbito científico e ingenieril se ha facilitado enormemente, gracias a la programación basada en directivas que proporciona OpenMP, debido a que se ahorra el desarrollo de código a bajo nivel, como puede ser el uso de hilos POSIX.

De hecho, actualmente `OpenMP` es tan popular, que programas como el AutoDock 4.2 que ayudan a encontrar una vacuna contra el SARS-CoV-2 (Norgan et al., 2011), simulando el proceso de acoplamiento molecular entre las células y el virus al predecir las interacciones de los receptores, utilizan `OpenMP` para acelerar los cálculos.

`OpenMP` puede considerarse como un estándar para la programación paralela de aplicaciones científicas sobre arquitecturas de memoria compartida, la lista de aplicaciones desarrolladas con `OpenMP` es basta y la lista de las más reconocidas se pueden consultar en `https://www.openmp.org/about/whos-using-openmp/`.

La clave de `OpenMP` para volverse tan popular, radica en que provee acceso a los sistemas paralelos de memoria compartida, sin un esfuerzo excesivo de programación. Por ejemplo, un bucle de control puede ser paralelizado insertando directivas, facilitando de este modo el paralelismo incremental, delegando la tarea de implementación del código a bajo nivel al compilador.

A pesar de las facilidades de paralelización por medio de directivas, que se basa en delegar al compilador generar el código paralelo, sin necesidad de hacer uso de llamadas a bajo nivel como Hilos POSIX, se debe entender que `OpenMP`, no es autosuficiente y aunque ayuda en gran medida el compilador con tareas de paralelización, no exime al programador de un buen diseño, ni de la planificación de las tareas. No obstante, el requerimiento de portabilidad de los programas ha conducido al uso extendido de `OpenMP` en máquinas de memoria compartida.

Cabe señalar que existen modelos de programación similares como: Cilk de Intel o HPF de IBM.

1.6.2.2 *Paso de Mensajes (MPI)*

El mecanismo de paso de mensajes consiste en un conjunto de tareas que tienen solamente memoria local, pero que pueden comunicarse con otras tareas enviando y recibiendo mensajes (ver Figura 1.4). La transferencia de datos de la memoria local de una tarea, a la memoria local de otra, es llevada a cabo por operaciones en ambas. El estándar más común es `MPI` (Message Passing Interface), y es una interfaz que define un conjunto de funciones para la comunicación y sincronización entre procesos, las funciones se integran en una librería, y aunque `MPI` no es propiamente una librería, la interfaz `MPI` se implementa como una librería y existen diferentes versiones como: OpenMPI, Intel MPI, MPICH MPI, LAM MPI, desarrolladas por distintas empresas u organizaciones de software libre.

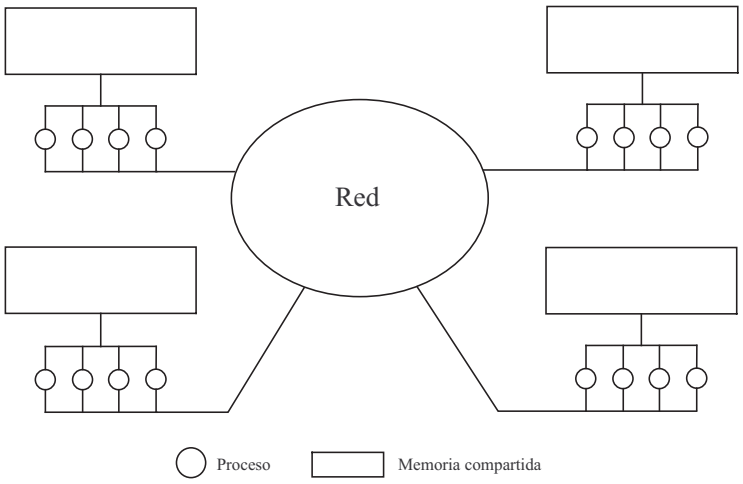


Figura 1.4: Topología MPI (Foster, 2020).

MPI prácticamente se ha convertido en el estándar de desarrollo en sistemas de memoria compartida, y es extremadamente portable, ya que los fabricantes conservan la especificación de la interfaz. En la mayoría de los casos los fabricantes optimizan una librería MPI para un equipo dado, por ejemplo, existen versiones de MPI optimizadas para máquinas de memoria compartida, basada en hilos de ejecución, debe tenerse en cuenta que en general todo modelo de memoria compartida puede ejecutarse en una máquina de memoria compartida, es decir, MPI funciona sin problemas en una arquitectura Multi-socket, Multi-núcleo.

MPI es la primera librería de paso de mensajes estándar y portable, especificada por consenso por el MPI Forum, con unas 40 organizaciones participantes, como modelo que permita desarrollar programas que puedan ser migrados a diferentes máquinas paralelas.

MPI es una especificación para programación de paso de mensajes, que proporciona una librería de funciones para `C`, `C++` o `FORTRAN` que son empleadas en los programas para comunicar datos entre procesos, pero no establece la topología de comunicación y prácticamente funciona sobre cualquier red, pero se recomienda una red segura de alta velocidad.

Otra implementación ya en desuso es la PVM (Parallel Virtual Machine) que permitía a un conjunto de computadoras tipo Unix comportarse como una sola máquina de memoria distribuida, esta red de computadoras se conoce como máquina virtual y se puede utilizar en varios niveles. En el nivel más alto, el modo transparente, las tareas se ejecutan automáticamente en la computadora más apropiada. En el modo dependiente de la arquitectura, el usuario especifica qué tipo de computadora ejecutará una tarea en particular. En el modo de bajo nivel, el usuario puede especificar una computadora en particular para ejecutar una tarea. En todos estos modos, PVM se ocupa de las conversiones de datos necesarias de computadora a computadora, así como de los problemas de comunicación de bajo nivel (Sunderam, 1990).

PVM es un sistema de transmisión de mensajes muy flexible. Admite la forma más general de cálculo paralelo MIMD. Esto permite el uso de cualquier paradigma de programación: todas las estructuras de control necesarias se pueden implementar con construcciones PVM adecuadas, no obstante, aunque esta disponible en el entorno de paquetes actuales de las diferentes distribuciones Linux, su uso es inusual en los sistemas de algo rendimiento.

1.6.2.3 *CUDA*

La tecnología CUDA basada en tarjetas gráficas fue introducida desde el 2007, y se ha convertido en una herramienta muy conveniente para desarrollar procesamiento numérico de alto rendimiento basado en unidades de procesamiento gráfico (GPU) (Dehal et al., 2018). El modelo de programación introducido para su manejo es CUDA C, el cual es una extensión a lenguaje C.

La tecnología CUDA no esta limitada a servidores con tarjetas de gama alta, con la gran cantidad de tarjetas gráficas instaladas en Laptops, equipos de escritorio, Workstations, es mucho más accesible y transportable. Por lo que el rango de las tarjetas CUDA puede ir desde la GeForce GTX 1650 para Laptops hasta las V200 de miles de dolares.

1.6.3 *La Arquitectura CUDA*

La arquitectura CUDA tiene los siguientes componentes:

- Las unidades de procesamiento gráfico (GPUs).
- El *driver* de software que permite el acceso al dispositivo.
- El ambiente de desarrollo a través de una extensión a lenguaje C.
- El conjunto de instrucciones PTX de la arquitectura (ISA). PTX proporciona un modelo de programación estable y un conjunto de instrucciones para la programación paralela de propósito general. Está diseñado para ser eficiente en las GPUs NVIDIA que soportan las funciones de cálculo definidas por la arquitectura NVIDIA Tesla (NVIDIA, 2010).

El ambiente de desarrollo, provee todas las herramientas, ejemplos y documentación necesaria para desarrollar aplicaciones que se benefician de la tecnología CUDA.

El ambiente de desarrollo esta compuesto por:

- librerías,
- ambiente de ejecución,
- herramientas,
- documentación,
- lenguaje de integración,
- interfaz a nivel de programación,
- y ejemplos.

Las librerías proporcionan un conjunto de librerías de funciones optimizadas para la plataforma CUDA, como por ejemplo, funciones de álgebra lineal (BLAS) o la transformada rápida de Fourier (FFT). El ambiente de ejecución (runtime), provee soporte para ejecutar las funciones estándar en el GPU utilizando el lenguaje C, para que las funciones puedan ejecutarse, es necesaria la comunicación entre el CPU y el GPU, esto se logra mediante el `driver` que permite el acceso al dispositivo.

Por la naturaleza del trabajo, lo más importante es el ambiente de desarrollo, compuesto básicamente del, runtime en C y las librerías.

El ambiente de desarrollo de CUDA, soporta dos interfaces de programación:

Una interfaz a nivel de programación de dispositivo. la cual es utilizada en la aplicación a través de DirectX, OpenCL o el CUDA API Driver para configurar el GPU, con la finalidad de lanzar los *kernels*, y retornar los resultados.

un lenguaje de integración como interfaz de programación. , en este caso se utiliza el Runtime de C y los desarrolladores utilizan una extensión a lenguaje `C` o `FORTRAN`, por medio de los cuales se específica que funciones (`kernels`) serán ejecutados en el GPU.

Las librerías que proporciona el ambiente están optimizadas para la arquitectura de los GPUs, como BLAS (van de Geijn and Goto, 2011). El ambiente de ejecución provee soporte para ejecutar las funciones (procedimientos) en C en el GPU, y los *bindings* para lenguajes como JAVA, FORTRAN y Python. Las herramientas que proporciona el ambiente de desarrollo son el compilador (nvcc), el depurador (cudagdb) y el perfilador visual (cudaprof).

La interfaz de programación con el dispositivo, es el `kernel`, la cual es un procedimiento o función que se replica concurrentemente en el GPU. El lenguaje tradicional para generar `kernels` es CUDA C, no obstante, otros lenguajes se pueden utilizar para generar los `kernels` como CUDA C.

Cuando se utiliza el lenguaje de integración, que casi siempre es CUDA C, se da por supuesto que esta configurado el GPU (correctamente instalado el driver), y permite la definición de funciones que se ejecutan en el GPU. Este tipo de programación permite sacar ventaja de lenguajes de mayor nivel como `C`, `C++`, `FORTRAN`, `JAVA`, `Python`, reduciendo la complejidad del código y los tiempos de desarrollo a través de la *integración de tipos* y la integración de código.

La *integración de tipos*, permite a los tipos de datos estándar, así como a los tipos de datos vectoriales, ser utilizados sin problemas en funciones ejecutadas en el GPU, es decir, los mismos tipos de datos que ocupamos en el CPU, los podemos ocupar en el GPU.

La *integración del código*, permite a la misma función ser llamada desde funciones que pueden ejecutadas desde el CPU y GPU.

Cuando es necesario distinguir funciones que serán ejecutadas en el GPU, se utilizan palabras clave de CUDA C, que permiten a los desarrolladores especificar que funciones serán ejecutadas en el GPU.

1.6.3.1 *Programación Híbrida MPI+OpenMP*

Las arquitecturas actuales son híbridas, los clusters actuales generalmente tienen de dos a cuatro procesadores por nodo, y cada procesador varios núcleos. Los nodos generalmente están están conectados por una red de alta velocidad como Infiniband, dada este tipo de arquitecturas, por lo tanto, es congruente que en el desarrollo de las aplicaciones paralelas actuales, se utilicen paradigmas híbridos de programación.

MPI en conjunto con OpenMP se han convertido en una herramienta extremadamente eficiente en los sistemas de cómputo paralelo e incluso en la computación de alto rendimiento (HPC) (Rabenseifner et al., 2009).

Principalmente, hay dos motivaciones para esta combinación de modelos de programación:

- Aumento de la capacidad de memoria distribuida.
- Se mejora el rendimiento debido a que se cuentan con más núcleos de procesamiento, por lo tanto es posible aumentar la escalabilidad.

Algunas aplicaciones están limitadas en los problemas que pueden resolver por la cantidad de datos que deben mantenerse en la memoria principal durante la ejecución. Las supercomputadoras modernas normalmente tienen al menos entre 128gb y 256 GBytes de memoria principal por nodo, y aunque instalar más memoria por nodo es técnicamente factible, los costos se elevan considerablemente, por lo que se convierte en inviable. Y aún cuando se dispusiera de una gran cantidad de memoria la aplicación deja de escalar debido a que los núcleos empiezan a interferirse entre sí por el manejo de memoria. Por esta razón la mejor manera de aumentar la memoria es de forma distribuida, es decir, uniendo varios nodos de procesamiento, no obstante, de esta manera el modelo de programación tiene que ser híbrido.

Cuando se utiliza MPI es común replicar parte de los datos que se procesan en cada nodo, es decir, un nodo n, replicara parte de sus datos en los nodos $n-1$ y $n+1$, estas regiones de replicación son conocidas como regiones sombra, donde cada proceso refleja los datos en el límite de su dominio computacional con sus procesos vecinos.

La pérdida de escalabilidad en MPI generalmente se va producir cuando la cantidad de información que se debe enviar por la red para comunicar cada nodo comienza a saturar el búferes de comunicación.

En una implementación híbrida MPI + OpenMP los datos se dividen en tareas que son mapeados a procesos y dentro del proceso, los datos son compartidos por subprocesos (OpenMP), por lo que se reduce la cantidad de regiones sombra. Una forma alternativa de ver este paradigma es dar a cada proceso de MPI suficiente memoria para ejecutar un problema determinado, y debido a que en cada nodo se distribuyen las tareas en forma de hilos, solo es necesario, ejecutar un solo proceso MPI por nodo. Esto da como resultado que se pueda explotar el paralelismo dentro de un proceso que utiliza subprocesos OpenMP y brinda la oportunidad de recuperar parte del rendimiento perdido asociado con tener muchos procesos MPI reduciendo las comunicaciones via red.

1.6.3.2 *MPI+OpenMP+CUDA*

Los sistemas de HPC basados en tarjetas gráficas, son cada vez más comunes, los GPU's han cambiado la forma de integrar los clusters, proporcionando un mayor poder de cómputo a un menor consumo energético, las unidades de procesamiento de gráficos (GPU) han evolucionado rápidamente para convertirse en aceleradores de alto rendimiento para la computación en paralelo basada en datos. Las GPU's modernas contienen cientos de unidades de procesamiento, capaces de lograr hasta 1 TeraFLOP (billón de operaciones de punto flotante por segundo) para aritmética de precisión simple (SP), y más de 80 GigaFLOPS (mil millones de operaciones de punto flotante por segundo) para precisión doble (DP) cálculos (Kindratenko et al., 2009).

Las GPU optimizadas para HPC contienen hasta 32 GB de memoria incorporada y son capaces de mantener anchos de banda de memoria superiores a 100 GB / seg. La arquitectura de hardware paralelo y el alto rendimiento de las operaciones de memoria y aritmética de punto flotante en las GPU las hacen especialmente adecuadas para cargas de trabajo científicas y de ingeniería que ocupan los clusters de HPC, lo que lleva a su incorporación como aceleradores de HPC.

Las GPU tienen el potencial de reducir significativamente las demandas de espacio, energía y refrigeración, y reducir la cantidad de imágenes del sistema operativo que deben administrarse en relación con los tradicionales clusters de solo CPU de capacidad computacional similar, de hecho NVIDIA ha comenzado a producir aceleradores de GPU llamados Tesla diseñados específicamente para clusters. Las GPU Tesla para HPC están disponibles como tarjetas adicionales estándar o en cajas de montaje en rack de 1U autónomas de alta densidad que contienen cuatro o más dispositivos GPU con alimentación y refrigeración independientes, esta última configuración es la adecuada para su conexión a nodos HPC montados en rack que carecen de espacio interno adecuado, energía o enfriamiento para instalación interna.

En el capítulo siguiente, siendo la arquitectura central de este trabajo, se abordará con detalle la arquitectura y programabilidad de los GPU's

1.7 ESTRATEGIAS PARA DESARROLLAR APLICACIONES PARALELAS

Para desarrollar una aplicación paralela surge la disyuntiva entre simplemente modificar una secuencial existente o desarrollar totalmente una nueva. En general, existen tres estrategias para desarrollar aplicaciones paralelas, como se muestra en la Figura 1.5.

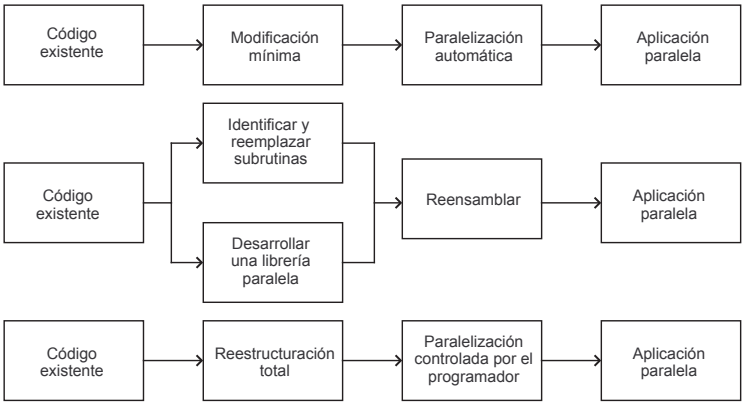


Figura 1.5: Estrategias para paralelizar aplicaciones paralelas.

La primera estrategia está basada en la paralelización automática y la segunda en el uso de librerías paralelas, mientras que la tercera estrategia consiste en un desarrollo y reestructuración total de la aplicación.

1.7.1 *Paralelización automática*

Existe investigación en compiladores de paralelización automática pero su funcionalidad todavía es muy limitada. El código generado ofrece un paralelismo regular basado en descomponer ciclos. Este tipo de compiladores conocidos como de vectorización/paralelización han probado su relativa eficacia en arquitecturas de memoria compartida y procesadores vectoriales, pero han fracasado rotundamente en sistemas de memoria distribuida como los clusters, aunque se desarrollaron herramientas como el OpenCluster de Intel Corp (descontinuado), que proporcionaron un rendimiento aceptable para algunas aplicaciones. La dificultad principal radica en que los clusters no tienen un tiempo de acceso uniforme a la memoria compartida.

En conclusión, la tecnología existente de compilación en la paralelización automática es limitada en alcance y provee un relativo factor de rendimiento aproximado de entre el 60 % y 70 %.

El objetivo de la paralelización automática es reducir el trabajo del programador. Un buen compilador acepta códigos sin una buena optimización y generará un código paralelo prácticamente sin esfuerzo adicional, sin embargo, ésta es una paralelización que deja mucho que desear, ya que es limitado el rendimiento que se puede obtener y depende mucho de la tecnología del compilador, además de que el control sobre la ejecución es reducido.

1.7.2 *Librerías paralelas*

La siguiente opción es modificar el código utilizando librerías paralelas. Esta aproximación, puede ser más eficiente que la anterior y la idea básica es sustituir el código secuencial por código contenido en las librerías paralelas. Las librerías se pueden utilizar de dos formas:

- encapsular el control de ejecución de la aplicación y
- proveer la implementación de rutinas que resuelven algún procedimiento en paralelo.

La desventaja de esta estrategia es que las librerías tienen un costo de adquisición o no se adecuan con precisión a nuestro problema, y aunque existen varias versiones libres, pueden ser una caja negra para el programador, en este modelo se pasan los parámetros como con cualquier otra función estándar y ésta se ejecuta en paralelo internamente. De los procedimientos contenidos en en la librería, no se tiene el control ni el conocimiento del como administra el paralelismo. Generalmente esta técnica demerita un poco el rendimiento porque no es un desarrollo integral del total de la aplicación y hereda el bajo rendimiento que se ha encontrado en las aplicaciones que se desarrollan utilizando componentes. Es recomendable utilizar librerías paralelas cuando se quiere paralelizar solo una parte del código sin una visión integral del mismo.

1.7.3 *Generación total de la aplicación paralela*

Esta estrategia involucra escribir la aplicación paralela desde el comienzo y decidir el diseño de manera global, proporciona una mayor libertad al programador quien puede escoger el lenguaje y el modelo de programación. No obstante, la tarea de elaboración es más difícil ya que prácticamente nada del código original puede ser reutilizado y aunado a un buen compilador se generan programas buenos en rendimiento y extensibilidad.

2

LA PLATAFORMA CUDA Y SU PARADIGMA DE PROGRAMACIÓN

La tecnología CUDA ha servido durante la última década como una herramienta factible para alcanzar el procesamiento numérico masivo basado en Unidades de Procesamiento Gráfico (GPU), que son los chips centrados de las tarjetas de video y una extensión a lenguaje C para la programación de propósito general. Con la diversificación de distintos tipos de tarjetas gráficas instaladas en Laptops, Desktops, Workstations y Cluster de Supercomputadoras, las aplicaciones que se ejecutan sobre la plataforma CUDA son bastante transportables. De hecho, las tarjetas gráficas con soporte para CUDA se pueden encontrar desde Laptops como la GeForce GTX 1650 hasta las honerosas V200 (Vuduc and Czechowski, 2011).

CUDA fue introducido en marzo del 2007 y con más de 100 millones de dispositivos vendidos a la fecha, miles de programadores utilizan las herramientas libre de CUDA para desarrollar aplicaciones de tipo científico y de usuarios domésticos, desde procesamiento de imágenes y video, simulaciones de procesos físicos, exploración de petróleo y gas, imágenes médicas e investigación científica.

En este capítulo se presenta el enfoque de la arquitectura y el paradigma de programación de los GPUs CUDA.

2.1 LA ARQUITECTURA CUDA

La arquitectura CUDA consiste de varios componentes que son:

- Los GPUs que contienen los chips que permiten el cómputo paralelo.
- El nivel de programación a través de funciones `kernel`, que se ejecutan en concurrencia en el GPU.
- El *driver* que permite el acceso al dispositivo y el API para desarrollo.
- El conjunto de instrucciones PTX de la arquitectura ISA (Instruction Set Architecture - Conjunto de Instruccines de la Arquitectura) (NVIDIA, 2010) para la computación paralela a través de `kernels` (funciones).

En este trabajo nos enfocaremos en el software de ambiente de desarrollo, el cual provee todas las herramientas, ejemplos y documentación necesaria para desarrollar aplicaciones que se benefician de la tecnología CUDA.

El ambiente de desarrollo esta compuesto básicamente de:

Librerías. Proporciona un conjunto de librerías que contienen BLAS (Basic Linear Algebra Subprograms) (van de Geijn and Goto, 2011), FFT (Fast Fourier Transform) , y otras funciones optimizadas para la arquitectura CUDA.

Runtime en c. El Runtime en C para CUDA provee soporte para ejecutar las funciones estándar en C en el GPU, y los *bindings* para lenguajes como FORTRAN, JAVA y Python.

Herramientas. Compilador C de NVIDIA (nvcc), Depurador de CUDA (cudagdb), Perfilador Visual CUDA (cudaprof), y otras herramientas de ayuda.

Documentación. Incluye la guía de Programación en CUDA, especificaciones API y otra documentación de ayuda relacionada.

Ejemplos. Ejemplos contenidos en el SDK (Standard Developer Kit) que demuestran buenas prácticas de programación para una gran variedad de Algoritmos en el GPU y aplicaciones.

El ambiente de desarrollo de CUDA, soporta dos interfaces de programación:

Una interfaz a nivel de programación de dispositivo. Esta es utilizada en la aplicación a través de DirectX, OpenCL o el CUDA API Driver para configurar el GPU, con la finalidad de lanzar los `kernels`, y retornar los resultados.

Un lenguaje de integración como interfaz de programación. En este caso se utiliza el Runtime de C y los desarrolladores utilizan una extensión a lenguaje `C` o `FORTRAN`, por medio de los cuales se específica las funciones (`kernels`) serán ejecutados en el GPU.

Cuando se utiliza la interfaz para la programación a nivel de dispositivo, los desarrolladores escriben los `kernels` contenidos en archivos separados en el lenguaje soportado por el API de su elección, por ejemplo, cuando se utilizan `kernels` con DirectX (compute shaders) están escritos en HLSL. Los kernels OpenCL están escritos en un lenguaje parecido a C, llamado OpenCL C. El API del Driver CUDA, acepta `kernels` escritos en C o PTX.

Cuando se utiliza el lenguaje de integración, que casi siempre es CUDA C, se da por supuesto que esta configurado el GPU (correctamente instalado el driver) y permite la definición de funciones que se ejecutan en el GPU. Este tipo de programación permite sacar ventaja de lenguajes de mayor nivel como `C`, `C++`, `FORTRAN`,

`JAVA`, `Python`, reduciendo la complejidad del código y los tiempos de desarrollo a través de la *integración de tipos* y la integración de código.

La *integración de tipos*, permite a los tipos de datos estándar, así como a los tipos de datos vectoriales, ser utilizados sin problemas en funciones ejecutadas en el GPU, es decir, los mismos tipos de datos que ocupamos en el CPU, los podemos ocupar en el GPU.

La *integración del código*, permite a la misma función ser llamada desde funciones que pueden ejecutadas desde el CPU y GPU.

Cuando es necesario distinguir funciones que serán ejecutadas en el GPU, se utilizan palabras clave de CUDA C, que permiten a los desarrolladores especificar que funciones serán ejecutadas en el GPU.

2.2 COMPUTACIÓN HETEROGÉNEA

En los inicios de la computación, las computadoras solo contenían una central de procesamiento (CPU) diseñada para correr tareas de propósito general, sin embargo, desde la última década, cada vez es más común que tengan otros elementos de procesamiento, y entre los más comunes son los GPUs. En un principio los GPU solo eran dedicados a mejorar el entorno gráfico, pero su poder de cómputo cada vez mayor, permitió que se convirtieran en herramientas de cálculo de propósito general, con un alto rendimiento y a un bajo costo energético, por lo cual, se han convertido en pieza fundamental de los más grandes equipos de supercómputo, sin embargo, los GPUs no pueden operar de forma autónoma, necesitan del CPU para su control, por lo que los CPUs y los GPUs están conectados por el bus PCI-Express, dentro de la misma placa base, en este tipo de arquitectura, los GPUs son referidos como dispositivos discretos (Ziabari et al., 2016).

La computación heterogénea se refiere al hecho de tener distintos tipos de dispositivos de cálculo integrados en una misma máquina o nodo, y aunque es una forma de aumentar el poder de cómputo, también incrementa enormemente la complejidad de la programación.

Una configuración heterogénea típica consiste de un CPU multi-núcleo, con uno hasta cuatro GPUs integrados en la misma placa.

Por la propia arquitectura heterogénea, una aplicación consiste de dos partes:

- Código Host (CPU)
- Código Device (Dispositivo GPU)

El código Host es el que se ejecuta en el CPU, y el código de dispositivo en el GPU, y en una aplicación heterogénea de este tipo el CPU es el responsable de manejar

el ambiente, el código y los datos que serán enviados al dispositivo para que realice tareas intensivas de cómputo.

El poder de cómputo que podemos obtener del dispositivo esta relacionado con sus capacidades de cómputo y memoria, y entre más capacidades tenga, es más costoso, las dos principales características que tiene un GPU son las siguientes:

- Número de núcleos CUDA
- El tamaño de memoria

y de estos se desprenden dos métricas que describen el rendimiento de un GPU.

- Rendimiento teórico computacional
- Ancho de banda de memoria

El rendimiento teórico computacional es una medida de la capacidad computacional, usualmente definida con base en el número de operaciones de punto flotante en precisión sencilla o doble que teóricamente el GPU puede procesar por segundo. El ancho de banda es una medida de la proporción en la que los datos se pueden leer o almacenar en la memoria. El ancho de banda de la memoria generalmente se expresa en GigaBytes por segundo (GB/s).

La Tabla 2.2 proporciona un breve resumen de la arquitectura de Fermi y Kepler y sus características de rendimiento.

Tabla 2.2: Comparación de las capacidades de una tarjeta Fermi vs Kepler. Características técnicas tomadas de la página de www.nvidia.com

	Fermi (Tesla C2050)	Kepler (TESLA K10)
CUDA Cores	448	2x1536
Memoria	6 GB	8 GB
Rendimiento Máximo	1.03 Tflops	4.58 Tflops
Ancho de Memoria	144 GB/s	320 Gb/s

Aunque la capacidades de cómputo de una GPU NVIDIA hacen referencia al número de núcleos y la cantidad de memoria que tiene, también se refiere al número de versión del equipo, que es el que nos permite saber que tipo de programación soporta, en la Tabla 2.3 se muestra la capacidad de cómputo con lo que se refiere a las características de programación soportadas.

Tabla 2.3: Comparación de las capacidades de programación de algunas tarjeta NVIDIA

GPU	Capacidad de Cómputo
GeFORCE RTX 3090	8.6
NVIDIA RTX 2060	7.5
Volta	7.2
NVIDIA TITAN XP	6.1
GeForce GTX 980 Ti	5.2

2.3 PARADIGMA DE LA COMPUTACIÓN HETEROGÉNEA

La computación en GPU no está encaminada a reemplazar la computación en el CPU. Cada paradigma tiene sus ventajas para cierto tipo de programas. La computación en CPU es recomendable para tareas de control intensivo, la computación en GPU es buena para el computación intensiva basada en datos, por lo tanto se complementan, y juntos producen una combinación poderosa. El CPU esta optimizado para cargas dinámicas de trabajo, marcadas por pequeñas secuencias de operaciones y flujo modificado por constantes bifurcaciones en la secuencia de ejecución del código, en contraste el GPU es distinto, las tareas deben ser dominadas por tareas con un control muy simple, es decir, prácticamente sin bifurcaciones (Lee et al., 2010).

Si un problema maneja un conjunto pequeño de datos, un sofisticado control lógico y un bajo nivel de paralelismo, el CPU es la mejor opción, debido a su capacidad para manejar instrucciones lógicas complejas y paralelismo a nivel de instrucciones. Si el problema en cuestión procesa una gran cantidad de datos y muestra un paralelismo masivo de datos, el GPU es la elección correcta porque tiene una gran cantidad de núcleos programables, puede admitir subprocesos múltiples masivos y tiene un ancho de banda máximo más grande en comparación con la CPU.

Las arquitecturas de computación paralela heterogénea de CPU + GPU evolucionaron porque la CPU y el GPU tienen atributos complementarios que permiten que las aplicaciones funcionen mejor utilizando ambos tipos de arquitecturas. Por lo tanto, para un rendimiento óptimo, es posible que deba utilizar tanto la CPU como el GPU para su aplicación, ejecutando las partes secuenciales o partes paralelas de tareas en la CPU y datos intensivos en paralelo en el GPU.

2.3.1 *Hilos CPU vs Hilos CUDA*

Los hilos o subprocesos en una CPU a pesar de que son considerados más ligeros que los procesos, son pesados si son comparados con los hilos de un GPU. El sistema

operativo debe administrar los hilos entre los núcleos de procesamiento, y conmutar entre hilos los canales de ejecución de la CPU para proporcionar un ambiente de multihilado, en este ambiente los hilos son atendidos por los distintos núcleos de la CPU. Los cambios de contexto son lentos y costosos, por eso aparecieron tecnologías como el HyperThreading que consiste en hacer esta conmutación rápida entre hilos.

Los hilos de las GPU son extremadamente ligeros y no son análogos a los hilos del CPU. En una configuración de aplicación típica de GPU, miles de los hilos se pueden crear y ponerse en cola para su ejecución. Si la GPU debe esperar por un grupo de hilos, simplemente comienza a ejecutar el trabajo en otro (Buck, 2007).

Los núcleos de CPU están diseñados para minimizar la latencia de uno o dos subprocesos a la vez, mientras que los núcleos de GPU están diseñados para manejar una gran cantidad de hilos para maximizar el rendimiento.

Hoy en día, una CPU con cuatro núcleos puede manejar de manera eficiente solo 4 hilos al mismo tiempo, u ocho si la CPU admiten HyperThreading. Las GPU NVIDIA modernas pueden admitir hasta 1536 subprocesos activos al mismo tiempo por multiprocesador. En GPU con 16 multiprocesadores, esto conduce a más de 24,000 hilos activos simultáneamente.

2.4 MODELO DE PROGRAMACIÓN CUDA

Los modelos de programación presentan una abstracción de arquitecturas de computadora que actúan como un puente entre una aplicación y su implementación en el hardware disponible. La Figura 2.6 muestra las capas importantes que se encuentran entre el programa y la implementación.

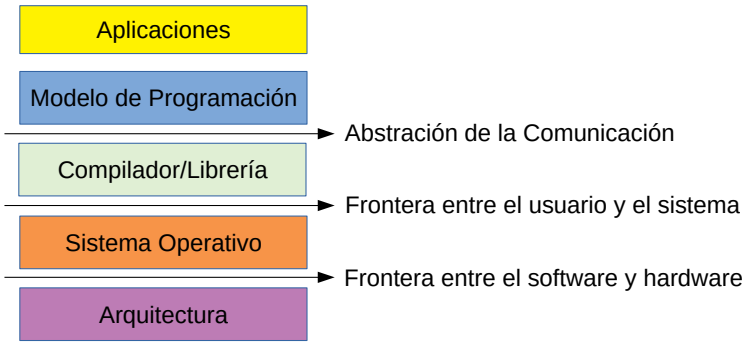


Figura 2.6: Capas de la plataforma.

La abstracción de la comunicación es la frontera entre el programa y la implementación del modelo de programación, que se realiza a través de un compilador o bibliotecas utilizando primitivas de hardware y el sistema operativo. El programa,

escrito para una programación modelo, dicta cómo los componentes del programa comparten información y coordinan sus actividades. El modelo de programación proporciona una vista lógica de arquitecturas informáticas específicas. Por lo general, es incrustado en un lenguaje de programación o su entorno.

Además de compartir varios conceptos abstractos con otros modelos de programación paralela, el modelo de programación CUDA proporciona las siguientes características especiales para aprovechar la potencia de cálculo del GPU.

- Una forma de organizar hilos en la GPU a través de una estructura de bloques.
- Un acceso jerárquico de la memoria en la GPU.

Desde la perspectiva de un programador, puede ver el cálculo paralelo desde diferentes niveles, como:

- Nivel de dominio.
- Nivel lógico.
- Nivel de hardware.

A medida que se trabaja en el diseño de un programa, esta actividad se centra en el nivel de dominio, que consiste en la forma de descomponer datos y funciones para resolver el problema de manera correcta y eficiente mientras se ejecuta en un entorno paralelo.

La fase de programación, se enfoca en la forma de organizar los hilos y la memoria, durante esta fase, se analiza el nivel lógico para garantizar que los hilos y los cálculos resuelven el problema correctamente, es análogo a la programación paralela en C, utilizando explícitamente Pthreads o técnicas OpenMP, por lo que CUDA expone una jerarquía de hilos para permitirle controlar su comportamiento.

Debe entenderse que la abstracción ofrece una escalabilidad superior para la programación paralela a nivel de hardware, y comprender cómo se asignan los hilos a los núcleos, puede ayudar a mejorar el rendimiento.

2.5 ESTRUCTURA DE PROGRAMACIÓN CUDA

El modelo de programación CUDA, permite ejecutar aplicaciones en sistemas heterogéneos (CPU+GPU), escribiendo código con un pequeño conjunto de extensiones al lenguaje de programación C.

Un entorno de desarrollo heterogéneo, consta de una CPU complementado con GPU's, cada una con su propia memoria, interconectados por el bus PCI-Express. Por lo tanto, se debe tener en cuenta la siguiente distinción:

- Host: la CPU y su memoria (memoria del Host).
- Dispositivo: la GPU y su memoria local (memoria del Dispositivo).

Una buena práctica de programación para ayudar a identificar claramente los diferentes espacios de memoria, consiste en ponerle prefijo a las variables o punteros que manejan la memoria utilizando nombres que comiencen con `h_` para la memoria del Host y `d_` para la memoria del Dispositivo.

A partir de CUDA 6, NVIDIA introdujo una mejora del modelo de programación llamada Memoria Unificada, que cierra la brecha entre los espacios de memoria del dispositivo y del Host. Esta mejora le permite acceder a la memoria de la CPU y la GPU con un solo puntero, mientras que el sistema migra automáticamente los datos entre el Host y el Dispositivo.

El punto medular de esta estrategia, es entender que se asigna memoria tanto del Host como en el dispositivo, y que se debe copiar explícitamente los datos que se comparten entre la CPU y la GPU. Este manejo de memoria es administrado por el programador y por lo tanto le brinda el poder de optimizar la aplicación y maximizar la utilización del hardware.

Un componente clave del modelo de programación CUDA es el `kernel` y el código que se ejecuta en la GPU. Como desarrollador, se tiene que tener en cuenta que un `kernel` es una función que se replica o se ejecuta en paralelo y que CUDA proporciona los mecanismos para gestionar los hilos en la GPU.

Desde el Host, se define cómo el algoritmo es mapeado al dispositivo basado en los datos de la aplicación y la capacidad computacional de la GPU. La intención es permitir concentrarse en la lógica del algoritmo de una manera sencilla, escribiendo funciones similares a lenguaje `C`, con la finalidad de evitar demasiados detalles con la creación y administración de miles de hilos en la GPU.

El Host puede funcionar independientemente del dispositivo para la mayoría de las operaciones. Cuando un `kernel` ha sido lanzado, el control se devuelve inmediatamente al Host, lo que libera a la CPU para realizar tareas adicionales complementado la ejecución paralela de datos que se ejecuta en el dispositivo. El modelo de programación CUDA es principalmente asíncrono para que el cálculo de la GPU realizado en la GPU se pueda superponer con comunicación Dispositivo-Host.

Un programa CUDA típico consta de un código de serial complementado por código paralelo, como se muestra en la Figura 2.7.

El código serial se ejecuta en el Host, mientras que el código paralelo se ejecuta en el dispositivo GPU. El código de Host está escrito en ANSI C, y el código del dispositivo está escrito usando CUDA C. Se puede poner todo el código en un solo archivo fuente, o puede utilizar varios archivos fuente para crear su aplicación o

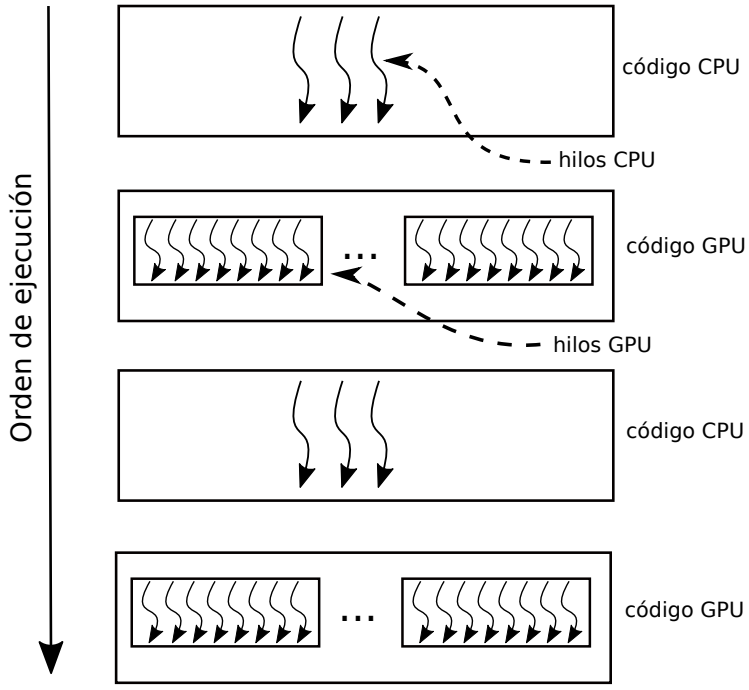


Figura 2.7: Secuencia de ejecución de un programa CUDA.

bibliotecas. El compilador NVIDIA C genera el código ejecutable tanto para el `Host` como para el dispositivo.

El flujo de procesamiento típico de un programa CUDA sigue este patrón:

- Copiar los datos de la memoria de la CPU a la memoria de la GPU.
- Invocar los `kernels` para operar con los datos almacenados en la memoria de la GPU.
- Copiar los datos de la memoria de la GPU a la memoria de la CPU.

2.5.1 *Manejo de memoria*

El modelo de programación CUDA asume un sistema compuesto por un Host y un dispositivo, cada uno con su propia memoria separada y los `kernels` funcionan en la memoria del dispositivo. Para permitir un control total y lograr el mejor rendimiento se deben de minimizar las transferencia de memoria entre el CPU y el GPU. CUDA proporciona funciones para asignar memoria en el Dispositivo, liberar la memoria del Dispositivo y transferir datos entre la memoria del Host y la del Dispositivo. La Tabla 2.4 enumera las funciones C estándar y sus funciones CUDA C correspondientes para operaciones de memoria.

Tabla 2.4: Host and Device Memory Functions

Funciones estándar en C	Funciones en CUDA C
malloc	cudaMalloc
memcpy	cudaMemcpy
memset	cudaMemset
free	cudaFree

La función utilizada para realizar la asignación de memoria de la GPU es `cudaMalloc` y su firma de función es:

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

La función utilizada para transferir datos entre el Host y el dispositivo es: `cudaMemcpy` y los parámetros son:

```
cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count,
                         cudaMemcpyKind kind )
```

Esta función copia los bytes especificados del área de memoria de origen, señalada por `src`, al área de memoria de destino, señalada por `dst`, con la dirección especificada por `kind`, donde `kind` toma uno de los siguientes tipos:

- `cudaMemcpyHostToHost`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`

Esta función exhibe un comportamiento sincrónico porque la aplicación de Host se bloquea hasta que `cudaMemcpy` regresa y la transferencia se completa. Todas las llamadas a CUDA, exceptuando las llamadas a los `kernel`, devuelven un código de error de un tipo enumerado `cudaError_t`. Por ejemplo, si la memoria del GPU es correctamente asignada devuelve, `cudaSuccess`. En otro caso, regresa `cudaErrorMemoryAllocation` .

Se puede convertir un código de error en un mensaje de cadena legible con la siguiente ejecución CUDA:

```
char* cudaGetErrorString(cudaError_t error)
```

La función `cudaGetErrorString` es análoga a la función `strerror`. El modelo de programación CUDA expone una abstracción jerárquica de la memoria del GPU, teniendo dos tipos de memorias: global y compartida.

Una de las características más notables del modelo de programación CUDA es la jerarquía de memoria explícita. Cada dispositivo de la GPU tiene un conjunto de diferentes tipos de memoria utilizado para diferentes propósitos en la jerarquía de memoria de la GPU, los dos tipos de memoria más importantes es la global y la compartida. La memoria global es análoga a la memoria del sistema de la CPU, mientras que la memoria compartida es similar a la caché de la CPU. Sin embargo, la memoria compartida de la GPU se puede controlar directamente desde un kernel CUDA C.

2.6 ORGANIZACIÓN DE LOS HILOS

Cuando se lanza un kernel desde el Host, la ejecución se traslada al dispositivo donde una gran cantidad de hilos son creados, y cada hilo ejecuta las declaraciones especificadas por el kernel. Saber cómo organizar los hilos es una parte fundamental de la programación en CUDA, y por lo tanto se proporciona una jerarquía que permite organizarlos para obtener el mejor rendimiento. Los hilos se organizan en dos niveles: a nivel de *grid* (malla) y de *blocks* (bloques).

Todos los hilos creados por el inicio de un solo kernel se denominan colectivamente *grid*, todos los hilos en un *grid* comparte la memoria global. Un *grid* está formado por muchos bloques de hilos. Un *bloque de hilos* es un grupo de hilos que pueden cooperar entre sí usando:

- Sincronización local de bloques.
- Sincronización de bloques en la memoria compartida.

Es importante señalar que los hilos de diferentes bloques no pueden cooperar.

Los subprocesos se basan en las siguientes dos coordenadas únicas para distinguirse entre sí:

- `blockIdx` (índice de bloque dentro de un grid).
- `threadIdx` (índice de hilo dentro de un bloque).

Estas coordenadas aparecen como variables preinicializadas integradas a las que se puede acceder dentro de la función del `kernel`. Cuando se ejecuta una función `kernel`, las variables de coordenadas son: `blockIdx` e `threadIdx` asignadas a cada hilo en tiempo de ejecución. Las coordenadas, permiten puede asignar partes de datos a diferentes hilos.

El tipo de la variable de coordenadas es `uint3`, un tipo de vector incorporado en CUDA, que representa una estructura que contiene tres enteros sin signo, los componentes dimensionales, son accesibles a través de los miembros `x`, `y` y `z` respectivamente.

```
blockIdx.x
blockIdx.y
blockIdx.z
threadIdx.x
threadIdx.y
threadIdx.z
```

CUDA organiza la malla y los bloques en tres dimensiones. En la Figura 2.8 muestra un ejemplo de un hilo estructura jerárquica con una malla 2D que contiene bloques 2D. Las dimensiones de una malla y un bloque son especificados por las siguientes dos variables integradas:

- blockDim (dimensión del bloque, medida en hilos)
- gridDim (dimensión de la cuadrícula, medida en bloques)

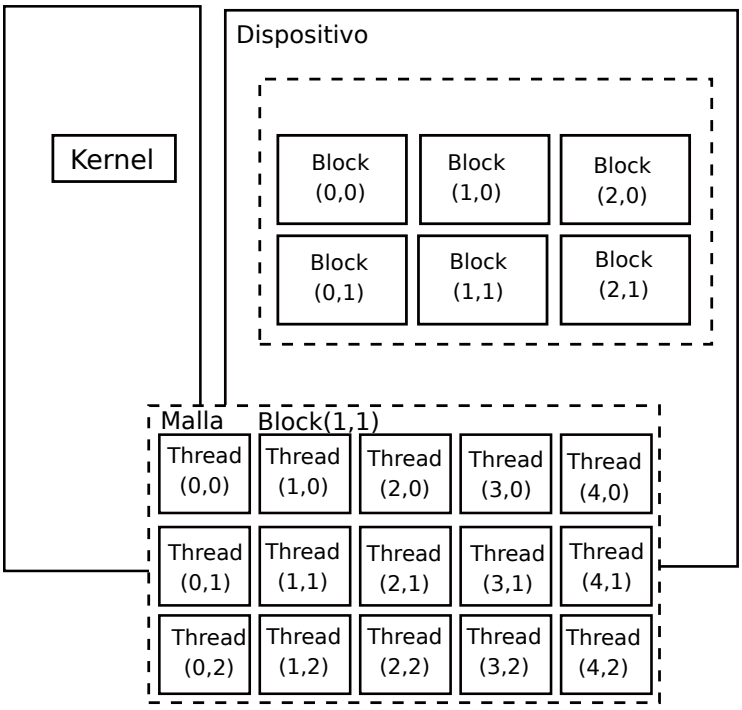


Figura 2.8: Estructura jerárquica de una malla 2D que contiene bloques 2D.

Estas variables son de tipo `dim3`, un tipo de vector entero basado en `uint3` que se usa para especificar dimensiones. Al definir una variable de tipo `dim3`, cualquier

componente que no se especifique se inicializa a 1. Cada componente de una variable de tipo `dim3` es accesible a través de sus campos `x`, `y` y `z`, respectivamente, como se muestra a continuación:

```
blockDim.x
blockDim.y
blockDim.z
```

2.7 CUDA COMO UNA PLATAFORMA HETEROGÉNEA DE CÁLCULO

CUDA es una plataforma de computación paralela de propósito general y un modelo de programación que aprovecha el motor de cómputo paralelo en las GPU's NVIDIA, con la finalidad de resolver muchos problemas computacionales complejos de una forma más eficiente y por lo tanto en mucho menos tiempo. Con CUDA, se puede acceder a la GPU para el cálculo numérico, como se ha hecho tradicionalmente en la CPU.

La plataforma CUDA es accesible a través de bibliotecas aceleradas por CUDA, directivas de compilación, aplicaciones e interfaces de programación y extensiones a lenguajes de programación estándar de la industria, incluyendo `C`, `C ++`, `FORTRAN` y `Python`.

CUDA C no solo es un conjunto de extensiones de lenguaje C para habilitar la programación heterogénea, también proporciona un conjunto de APIs para administrar dispositivos, memoria y otras tareas, por lo tanto, CUDA puede considerarse como un modelo de programación escalable que permite a los programas escalar de manera transparente su paralelismo con las GPU con diferentes números de núcleos, manteniendo un aprendizaje relativamente fácil con el lenguaje de programación C.

CUDA proporciona dos niveles de API para administrar el GPU y organizar subprocesos:

- CUDA Driver API (API del controlador)
- CUDA Runtime API (API en tiempo de ejecución)

El API del controlador es una API de bajo nivel y es relativamente difícil de programar, pero proporciona más control sobre cómo se usa el dispositivo GPU. La API de tiempo de ejecución es una API de nivel superior implementada por encima del Driver. Cada función contenida en el Runtime API se descompone en funciones más básicas contenidas en el API Driver.

No hay una diferencia de rendimiento notable entre utilizar el API del driver o del tiempo de ejecución. El rendimiento se obtiene al manejar eficientemente los `kernels`, la memoria, y la forma en como se organizan los hilos CUDA dentro de la

función, todo esto en conjunto tiene un efecto mucho más notorio en el rendimiento, es necesario señalar que las dos APIs se excluyen mutuamente y se debe utilizar una u otra, pero no es posible mezclar llamadas a funciones de ambas.

Un programa CUDA consta de una combinación de las siguientes dos partes:

- El código Host se ejecuta en la CPU.
- El código de dispositivo se ejecuta en la GPU.

La instrucción para invocar el compilador CUDA a través de la terminal para diferentes plataformas es `nvcc` y utiliza el compilador de C como base, y separa el código del dispositivo del código del Host durante la compilación. El código del dispositivo se escribe utilizando un lenguaje C extendido con palabras clave para etiquetar funciones paralelas de datos, llamadas *kernels*. El código del dispositivo se compila a través del comando `nvcc`. Durante la etapa de enlace en la compilación, el `nvcc` agrega las bibliotecas contenidas en el API del runtime de CUDA para llamadas a procedimientos del `kernel` y funciones explícitas de manipulación.

El compilador NVCC se basa en la infraestructura del compilador de código abierto LLVM ampliamente utilizada (`https://llvm.org/`). Puede crear o ampliar lenguajes de programación con soporte para aceleración de la GPU usando el SDK del compilador CUDA.

2.8 CONSIDERACIONES SOBRE LA PLATAFORMA CUDA

Se debe considerar que no todo tipo de problemas computacionales 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 secuencia de código a todos los datos de entrada. Se puede decir que una solución de un problema en la GPU será más ventajosa respecto a la solución en la CPU si la aplicación tiene las siguientes propiedades:

- No existe una necesidad de la aplicación de estar transfiriendo información entre la CPU y la GPU.
- La carga de cálculo computacional en cada hilo, debe ser lo suficientemente grande de tal forma que compense el tiempo de transferencia de información.
- No debe existir dependencia entre los datos para realizar los cálculos, esto es posible si cada SM (Streaming Mul-tiprocessor), 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.
- Se debe minimizar la transferencia de información entre la GPU y la CPU. La situación óptima es cuando la transferencia sólo se realiza una vez, al co-

mienzo y al final del proceso. No obstante, siempre se requieren transferencias intermedias debido a que se requieren resultados parciales.

- Se debe evitar el uso de secciones críticas, es decir, se debe evitar que varios hilos escriban 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 el acceso a un recurso compartido, luego entonces se convierten en una región crítica, lo cual implica contar con mecanismo de acceso seguro lo que implica ralentizar la solución del proceso global.

Además es necesario, que las estructuras de datos sean de tipo numérico y puedan transformarse a estructuras de tipo matriz o vector con la finalidad de poder manejarlas en 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 del procesador principal (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, y 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 definir 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 dispositivo (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` 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. El `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.

2.9 MODELO DE PROGRAMACIÓN CUDA

CUDA propone un modelo de programación SIMD (Instrucción Simple-Múltiples Datos) con funcionalidades de procesamiento de tipo vectorial. 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 determinado 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.

Como se dijo anteriormente 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.

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.

Hasta aquí se abarcaron los tópicos más sobresalientes que tiene la programación en CUDA, no obstante, la tecnología es muy extensa, y no es el propósito de este trabajo adentrarse con detalle en todo lo que CUDA proporciona, de hecho hay libros y manuales al respecto, no obstante, es necesario introducir los conceptos básicos del uso de la plataforma, para contextualizar la estructura propuesta.

3

DISEÑO E IMPLEMENTACIÓN DE ESTRUCTURAS DE DATOS BIDIMENSIONALES Y TRIDIMENSIONALES EN CUDA C

En este capítulo introducirá la estructura propuesta, para poder manejar dos o tres índices dentro de funciones kernel como en C estándar, con el objetivo de facilitar el mantenimiento y programabilidad del código.

3.1 ESTRUCTURA BIDIMENSIONAL

Para introducir la creación de arreglos 2D en CUDA, es decir, que ambos índices `[][]` puedan ser utilizados dentro de un `kernel`, se tomará como base la forma tradicional de crear un arreglo 2D en C estándar usando memoria contigüa (El código es mostrado en el Listado 3), es conveniente utilizar el tipo `float`, pero cualquier tipo primitivo puede ser utilizado.

Listado 3: Asignación de memoria continua para un arreglo 2D en C.

```
float ** GETMEMORYCONTINUA_MATRIX_C_F(int n, int m){
float ** A = NULL;
int i,j;

cudaMallocHost((void***)&A,n*sizeof(float*));

if (A == NULL)
puts("Error cudaMallocHost first level"),
exit (-1);

cudaMallocHost((void**)&A[0],n*m*sizeof(float));

if (A[0] == NULL) puts("Memory second level"), exit (-1);

for (i=1; i<n; i++)
  A[i] = A[0] + i * m;

return A;
}
```

La función mostrada en el Listado 3 es una asignación continua de memoria en la CPU, esto es muy importante ya que permite manejar un arreglo bidimensional como unidimensional de manera continua y se usará como estructura principal para crear una matriz 2D en CUDA, como se muestra en el Listado 4.

En la Figura 3.9 se muestra la estructura contigua que se necesita en la memoria para representar un arreglo 2D, que consiste en pedir memoria para un arreglo unidimensional pero asignar los correspondientes punteros en las localidades cada `m` columnas y de esta crear un arreglo 2D renglonizado continuamente.

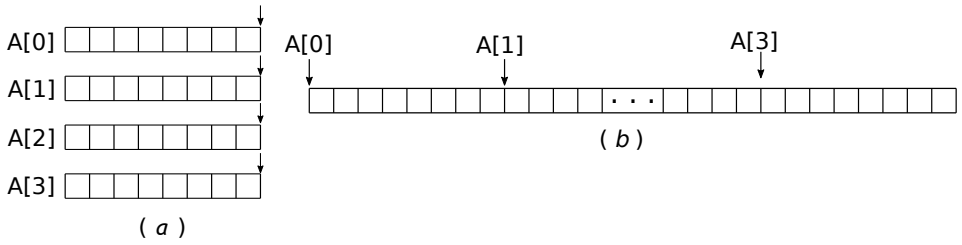


Figura 3.9: Estructura de memoria continua bidimensional en la CPU. (a) Arreglo 2D sin continuidad, (b) Arreglo 2D contiguo.

En el Listado 4, la función `MATRIX_GPU` tiene 4 parámetros: un puntero doble el cual recibe el puntero que nos permite hacer el manejo de la memoria entre el CPU y el GPU, y un puntero triple que recibe el puntero a puntero que contiene el arreglo bidimensional que puede ser utilizado dentro del `kernel`, y los enteros `m` y `n` que definen el tamaño del arreglo bidimensional.

El puntero `**(P)` es utilizado para recibir la dirección del puntero que apunta al arreglo en su forma contigüa, y que es utilizado para transferir la información entre el CPU el GPU, el puntero triple `***(M)` recibe la dirección recibe la dirección del puntero el cual será utilizado dentro de los `kernels` con los índices `[][]`, `n` y `m` son el número de renglones y columnas respectivamente.

Listado 4: Asignación de memoria continua para un arreglo 2D en CUDA

```
void GETMEMORYCONTINUA_MATRIX_GPU(float **P,float ***M,int n,
    int m)
{
 int i;
 float ** P_M, ** dev_M;
 P_M = (float **) malloc(n*sizeof(float*));
 if(P_M == NULL){
  printf("\nMemory error");
  exit(-1);
 }
 cudaMalloc((void**)&P_M[0],n*m*sizeof(float));
```

```
 if(P_M[0]==NULL){
  printf("\nMemory error");
  exit(-1);
 }
 for (i=1; i<n; i++)
  P_M[i] = P_M[0] + i * m;

 cudaMalloc((void***)&dev_M,n*sizeof(float*));
 if(dev_M==NULL){
  printf("\nMemory error");
  exit(-1);
 }
 cudaMemcpy(dev_M,P_M,n*sizeof(float*),
  cudaMemcpyHostToDevice);
 *(P) = P_M[0];
 *(M) = dev_M;
}
```

El código mostrado en el Listado 4 cuenta con dos punteros importantes: `P_M` y `dev_M`. El primero sirve para transferir los datos entre la CPU y la GPU, y el segundo es utilizado para manejar el arreglo dentro de un `kernel`, el hecho de tener 2 punteros uno para el manejo y otro para la operación es lo que permite tener una estructura bidimensional dentro del `kernel`. La idea principal para la creación de un arreglo arreglo 2D en CUDA de tamaño `m x n` se resume como sigue:

- Crear un arreglo de punteros de primer nivel de `n` elementos de tipo `float *` apuntados por el puntero doble `P_M` en la CPU.
- En el primer elemento del arreglo `P_M[0]`, se asigna memoria para `m x n` elementos de tipo `float` en la GPU.
- Asignar a cada elemento de `P_M[i]` de 1 hasta `n`, la dirección de memoria `P_M[0] + m x i`, creando de esta forma el arreglo 2D.
- Crear un arreglo de punteros de primer nivel de `n` elementos de tipo `float *` apuntados por el puntero doble `dev_M` en la GPU.
- Transferir las direcciones de memoria contenidas a partir del puntero `P_M[i]` al GPU, lo que implica copiar el contenido del puntero `P_M` a `dev_M`, donde `dev_M` puede ser utilizado dentro de un `kernel` CUDA como un arreglo 2D.

Este último punto es de suma importancia debido a, que es el que permite manejar los índices bidimensionalmente, y la clave consiste en transferir no solo los valores, sino, también las direcciones memoria de los punteros que apuntan a cada renglón. En la Figura 3.10 se muestran en verde los valores del arreglo que son transferidos de la CPU a la GPU y en amarillo las direcciones de memoria.

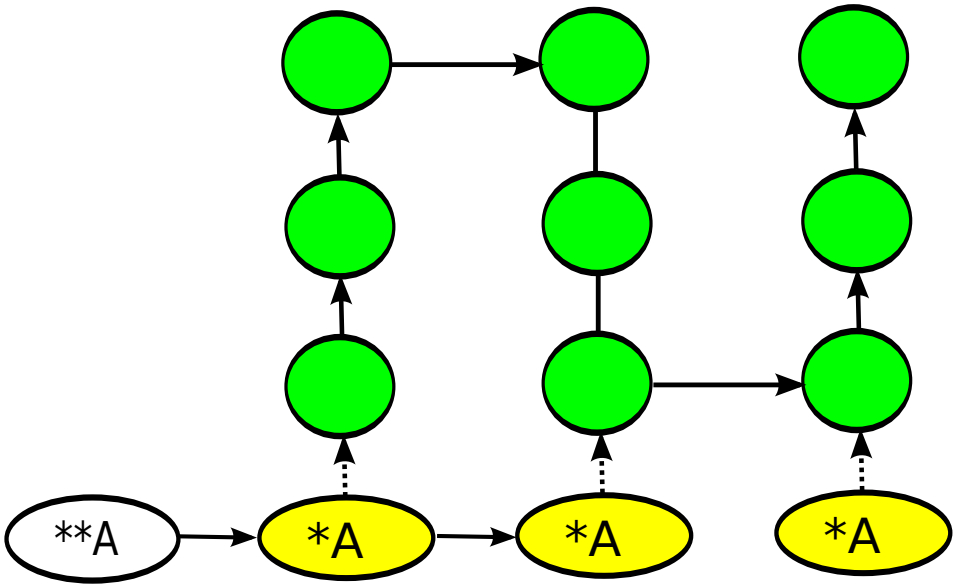


Figura 3.10: Asignación de memoria continua para un arreglo 2D

3.2 ESTRUCTURA TRIDIMENSIONAL

Análogamente a la creación de arreglos de memoria continua en 2D, en el caso 3D es necesario asignar memoria contigua. En el Listado 5, se asigna memoria continua para un arreglo 3D en la memoria de la CPU. Tomando este código como base, se puede construir su equivalente en CUDA mostrado en el Listado 6.

Similarmente a la estructura utilizada para el arreglo bidimensional en la GPU, definimos una función `CUBE_GPU` de cinco parámetros: dos punteros y tres enteros, donde el puntero a puntero `P_M` es utilizado para crear una estructura que permita transferir el contenido del arreglo 3D entre la CPU y la GPU. El segundo puntero `dev_M`, es un puntero de tipo `****`, y funciona para manejar el arreglo 3D dentro del `kernel`. Los tres enteros `m`, `n` and `z` definen el tamaño del arreglo tridimensional, por lo tanto la estructura queda de tamaño `m x n x z`. El detalle de su implementación procede como sigue:

- Crear un puntero de tercer nivel (`P_M`) para almacenar `n` direcciones de segundo nivel en la CPU.
- En la primera localidad de `P_M[0]`, asignar memoria para almacenar `m x n` direcciones de primer nivel en la CPU.
- Asignar a cada elemento de `P_M[i]`, la dirección de memoria de la localidad del elemento `P_M[0] + m x i`.

- Utilizar el puntero `P_M[0][0]`, para crear un arreglo lineal de tipo `(float)` de tamaño `n x m x z` en la GPU.
- Crear un puntero de tercer nivel (`dev_M`) para almacenar `n` direcciones de segundo nivel en la GPU.
- Crear un puntero de segundo nivel (`dev_M_2D`) para almacenar `n x m` direcciones de primer nivel en la GPU.
- Transferir las `n x m` direcciones contenidas en `P_M[0]` a `dev_M_2D` de tipo `float *`.
- Transferir las `n` direcciones de tipo `float **` de `P_M` a `dev_M`.
- Asignar `*(P) = P_M[0][0]`.
- Asignar `*(M) = dev_M`

Listado 5: Asignación continua de memoria en C para un arreglo 3D.

```
float *** CUBE_CPU(int n, int m, int z)
{
 float *** A = NULL;
 float *p = NULL;
 int i,j,k;
 cudaMallocHost((void****)&A,n*sizeof(float**));
 if (A==NULL)
 {
  printf("\nMemory problem");
  exit(-1);
 }
 cudaMallocHost((void***)&A[0],n*m*sizeof(float*));
 if(A[0]==NULL){
  printf("\nMemory problem");
  exit(-1);
 }
 for(i=1; i<n; i++)
  A[i] = A[0] + i*m;
 cudaMallocHost((void**)&A[0][0],n*m*z*sizeof(float));
 if(A[0][0] == NULL){
  printf("\nMemory problem");
  exit(-1);
 }
 for(j=1; j<(n*m); j++)
  A[0][j] = A[0][0] + j*z;
 return A;
}
```

Listado 6: Asignación de memoria continua en C para un arreglo 3D en el GPU.

```
void CUBE_GPU(float ** P,float **** M,int n,int m,int z)
{
 int i, j;
 float *** P_M = NULL, *** dev_M = NULL;
 float ** dev_M_2D = NULL;

 cudaMallocHost((void****)&P_M,n*sizeof(float**));

 if(P_M == NULL){
  printf("\nError Host Pointers");
  exit(0);
 }

 cudaMallocHost((void***)&P_M[0],n*m*sizeof(float*));

 if(P_M[0]==NULL){
  printf("\nError Host Pointers");
  exit(0);
 }

 for (i=1; i<n; i++)
  P_M[i] = P_M[0] + i * m;

 cudaMalloc((void**)&P_M[0][0],n*m*z*sizeof(float));

 for (j=1; j<(n*m); j++)
  P_M[0][j] = P_M[0][0] + j * z;

 cudaMalloc((void****)&dev_M,n * sizeof(float **));
 cudaMalloc((void***)&dev_M_2D,n * m * sizeof(float*));

 if(dev_M== NULL)
 {
  printf("\nError dev Pointers 3D");
  exit(0);
 }
 if(dev_M_2D==NULL)
 {
  printf("\nError dev Pointers 2D");
  exit(0);
 }
 cudaMemcpy(dev_M_2D,P_M[0],n*m*sizeof(float*),\
    cudaMemcpyHostToDevice);

 *(P) = P_M[0][0];
```

```
 for (i=0; i<n; i++)
  P_M[i] = dev_M_2D + i * m;
48
 cudaMemcpy(dev_M,P_M,n*sizeof(float**),\
    cudaMemcpyHostToDevice);

 *(M) = dev_M;
}
```

4

APLICACIÓN A LA ECUACIÓN DE TRANSFERENCIA DE CALOR

En este capítulo se presenta una aplicación de la estructura propuesta en este trabajo, que consiste en resolver la ecuación de difusión de calor en 2D y 3D, por medio de diferencias finitas con un esquema explícito en tiempo.

4.1 LA ECUACIÓN DE TRANSFERENCIA DE CALOR

La ecuación que describe la conducción de calor en sólidos ha demostrado, durante los dos últimos siglos, ser una poderosa herramienta para analizar el movimiento dinámico del calor, así como para resolver una enorme variedad de problemas de difusión en ciencias físicas, ciencias biológicas, ciencias de la tierra y ciencias sociales. Esta ecuación fue formulada a principios del siglo XIX por uno de los eruditos más talentosos de ciencia moderna, Joseph Fourier de Francia (Narasimhan, 1999).

La ecuación de calor es una ecuación diferencial parcial (PDE), la cual describe la distribución de calor (o variación de temperatura) en una región determinada a lo largo del tiempo. Es muy importante entender la diferencia entre calor y temperatura, el calor es un proceso de transferencia de energía como resultado de la diferencia de temperatura entre dos puntos. Por tanto, el término *calor* se utiliza para describir la energía transferida a través del proceso de calentamiento. La temperatura, por otro lado, es un factor físico propiedad de la materia que describe el calor o la frialdad de un objeto o ambiente, por tanto, no se intercambiaría calor entre cuerpos de la misma temperatura (Pletcher et al., 2012).

Si suponemos que tenemos una función u, que depende de las componentes espaciales y el tiempo (x, y, z, t), que describe la temperatura de un conductor material en una ubicación determinada (x, y, z), y podemos utilizar esta función para determinar la temperatura en cualquier posición del material en el tiempo $t + 1$ (tiempo siguiente), por lo tanto, los valores de la función u cambian conforme progresa el tiempo, y la ecuación de calor se utiliza para determinar este cambio en la función u.

El gradiente de u describe la dirección y velocidad a la que la temperatura se difunde en una región particular del material, por lo tanto, el gradiente de temperatura es el flujo de calor a través del material. Este gradiente, nos ayudará a

determinar el flujo de calor a través de diversos materiales. Esto es análogo al flujo de agua en una tubería.

Sea $U(x, y, z, t)$ una función en el espacio cartesiano (x, y, z) y t como la variable en el tiempo, la ecuación de calor se define como:

$$\frac{\partial U}{\partial t} = \alpha \left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + \frac{\partial^2 U}{\partial z^2} \right) \tag{1}$$

más generalmente,

$$\frac{\partial U}{\partial t} - \alpha \nabla^2 U = 0, \tag{2}$$

donde α es una constante positiva y es la difusividad termal, ∇^2 denota el operador de Laplace, y $U(x, y, z, t)$ denota la variación de la temperatura en el espacio-tiempo, para el presente trabajo se toma $\alpha = 0.5$.

La ecuación de transferencia de calor es de fundamental importancia en diversos campos científicos y es una ecuación diferencial parabólica. En teoría de probabilidad, la ecuación de calor esta conectada con el estudio del movimiento Browniano vía la ecuación de Fokker-Planck. En matemáticas financieras es utilizada para resolver la ecuación diferencial de Black-Scholes. Pero generalmente es utilizada para resolver difusiones químicas y procesos relacionados.

Adicionalmente, la ecuación de calor se usa en probabilidad y describe trayectorias aleatorias, se aplica en matemáticas financieras y también es importante en la geometría Riemanniana y, por lo tanto, en la topología, de hecho la ecuación fue adaptada por Richard S. Hamilton cuando definió el flujo de Ricci que luego fue utilizado por Grigori Perelman para resolver la conjetura topológica de Poincaré.

Para los propósitos de este trabajo determinaremos el flujo de calor en una placa y un cubo, que se asemeja a la forma de un satélite CubeSat (Piedra et al., 2019), y aunque no es el objetivo de este trabajo, puede servir de base para el diseño y la validación de un susbsistema térmico para un nanosatélite de 1U.

4.2 LA ECUACIÓN DE TRANSFERENCIA DE CALOR EN DIFERENCIAS FINITAS

El método de diferencias finitas (FDM) es un método de aproximación para resolver ecuaciones diferenciales parciales, ha sido utilizado para resolver un amplio rango de problemas (Forsythe and Wasow, 1960). Estos incluyen problemas lineales y no lineales, independientes del tiempo y dependientes. Este método se puede aplicar a problemas con diferentes formas de contorno, diferentes tipos de condiciones de frontera y para una región que contiene varios materiales diferentes. Aunque el

método era conocido por Gauss y Boltzmann, no se usaba ampliamente para resolver problemas de ingeniería hasta la década de 1940.

Similar a otros métodos numéricos, el objetivo de la diferencias finitas es reemplazar un problema de campo continuo con infinitos grados de libertad por un campo discretizado con nodos regulares finitos. Las derivadas parciales de la función desconocida son aproximadas por los cocientes en diferencias en un conjunto de puntos de discretización finitos. La ecuación diferencial parcial original se transforma luego en un conjunto de valores algebraicos y la solución de estas ecuaciones es la solución aproximada al valor límite original.

La idea básica de FDM es reemplazar las derivadas de una función desconocida por los cocientes en diferencias de funciones desconocidas. La forma en diferencias finitas de las ecuaciones depende de la forma de discretización del dominio. Si asumimos un área bidimensional Ω limitada por el contorno Γ, la función potencial U dentro del dominio Ω satisface la ecuación de Poisson y está sujeto a las condiciones de Dirichlet como se muestra a continuación:

$$\begin{aligned} \nabla^2 &= F(x, y), \text{ en el dominio } \Omega \\ U|_\Omega &= g(\Omega). \text{ a la frontera } \Gamma \end{aligned} \tag{3}$$

En principio Ω puede ser dividido en una malla arbitraria como se muestra en la Figura 4.11.

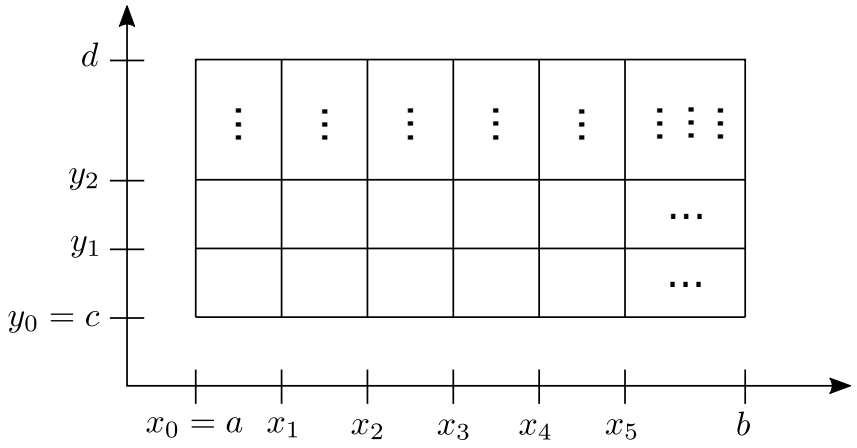


Figura 4.11: Dominio discreto en 2D.

Es necesario señalar que existen diferentes métodos basados en diferencias finitas, implícitos, explícitos, compactos. No obstante, utilizaremos el método relativamente más sencillo, ya que el propósito de este trabajo es demostrar la utilidad de la estructura de datos y no profundizar sobre los distintos esquemas de diferencias finitas bien establecidos para la ecuación de transferencia de calor.

Para tal efecto, comenzaremos analizando la ecuación de transferencia de calor unidimensional, escrita como sigue:

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} \right). \tag{4}$$

Físicamente podemos pensar en el dominio unidimensional como un cable de longitud x, donde se ignora la dimensionalidad, y tiene condiciones de temperatura constantes en los extremos (condiciones de frontera) y se debe especificar la temperatura en cada punto discreto x_n.

Para resolver este problema unidimensional, necesitamos transformar la ecuación (4), utilizando una diferencia finita central de segundo orden para el espacio (x) y una diferencia hacia adelante en el tiempo con la finalidad de simplificar la programación, otra aproximación para el tiempo implicaría la resolución de sistemas de ecuaciones tridiagonales. Por lo tanto la ecuación (4) es convertida a una forma algebraica como (Zhou, 1993):

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha \left(\frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2} \right) \tag{5}$$

donde u es una función discreta que depende de (i, t), i es el índice espacial, y t el espacial, las condiciones a la frontera son impuestas en $i = 0$ y $i = n$, donde n representa el número de puntos discretos totales (ver Figura 4.12).

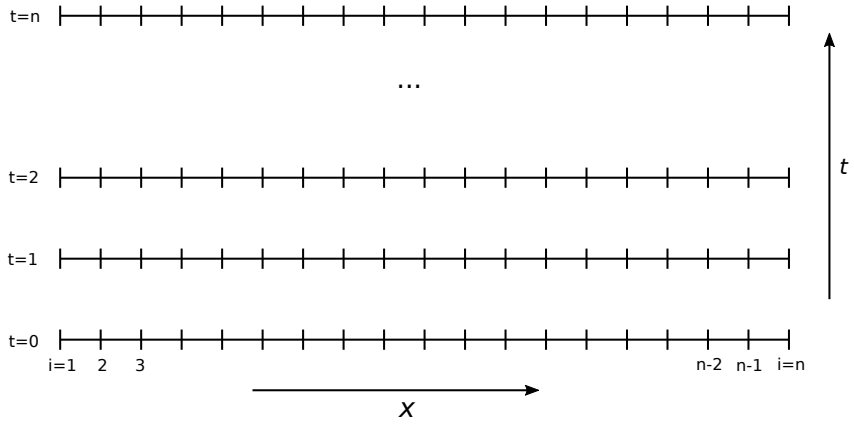


Figura 4.12: Solución discreta de la ecuación 1D.

Despejando u_i^{t+1}, rearreglamos la ecuación (5) como:

$$u_i^{n+1} = \frac{\alpha \Delta t}{\Delta x^2} \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n \right), \tag{6}$$

a esta ecuación se le conoce como explícita en tiempo y centrada en el espacio, y agrupando términos semejantes queda de la siguiente manera:

$$U_i^{n+1} = rU_{i+1}^n + (1-2r)U_i^n + rU_{i-1}^n, \tag{7}$$

donde $r = \dfrac{\alpha\Delta t}{\Delta x^2}$.

El esquema explícito en tiempo es relativamente fácil de implementar debido a que los valores de U_i^{n+1} pueden ser actualizados, independientemente de otros, por lo que la solución se puede implementar con dos *ciclos*, el ciclo exterior, que corresponde al tiempo y el ciclo interior corresponde a los nodos espaciales interiores.

Uno de los inconvenientes de la formulación explícita en tiempo es que se convierte en inestable si Δt es demasiado grande, por lo que se debe cumplir el siguiente criterio para conservar la estabilidad.

$$\frac{\alpha\Delta t}{\Delta x^2} < \frac{1}{2}. \tag{8}$$

4.3 CASOS DE APLICACIÓN

4.3.1 *Suma de Matrices*

Para introducir a la legibilidad de la estructura desarrollada en este trabajo, primeros se debe construir un programa para sumar dos matrices A y B, almacenando el resultado en C, la manera tradicional de hacer esta operación en CUDA `C` es manejando las matrices como arreglos unidimensionales, lo que implica un solo índice, para llevar a cabo la operación se necesita almacenar de forma unidimensional la matrices, si `a` es un puntero a los elementos de la matriz `*a` y se desea crear una matriz a partir de este puntero en la memoria de la GPU, entonces utilizamos `cudaMallocHost` de la siguiente manera: `a=cudaMallocHost((void*)a,n*m*sizeof(float))`, de tal forma que si tenemos un iterador `i` sobre `n` y `j` sobre `m`, el acceso será de la forma `i*m+j`, esta forma unidimensional de crear y manejar matrices se muestra en el Listado 7.

Listado 7: Estructura bidimensional mapeada a un arreglo unidimensional.

```
float *a;
a=cudaMallocHost((void*)a,n*m*sizeof(float));

for(i=0;i<n;i++)
 for(j=0;j<m;j++)
  a[i*m+j] = float(i+j);
}
```

Y aunque es la forma común de manejar arreglos multi-dimensionales en CUDA, puede ser propenso a errores, debido a la complejidad de los algoritmos utilizados, como la solución de Ecuaciones Diferenciales Parciales, es necesario señalar que una de las herramientas que provee el CUDA API es la función `CudaMallocPitch(...)` la cual permite crear arreglos 2D. El uso de `CudaMallocPitch(...)` se muestra en el Listado 8, donde se usa para el caso de un arreglo de 3×3 como sigue:

Primero, asignamos memoria utilizando el puntero `**g_a` `cudaMallocPitch(&g_a, &pitch1, 10*sizeof(float), 10);` y posteriormente transferimos el contenido de la memoria del host al dispositivo, `cudaMemcpy2D(g_a,pitch1,a,SIZE*sizeof(float), 3*sizeof(float),3,cudaMemcpyHostToDevice)`. Hasta este punto su uso parece simple, sin embargo, cuando el kernel es ejecutado, debemos extraer cada renglón de la matriz, `float*rowa=(float*)((char*)gpu_a+idx*pitch1)` y entonces cada elemento del arreglo es manejado como: `rowc[idy] = rowa[idy] + rowb[idy]`, y aunque al momento de pedir memoria se maneja como un arreglo bidimensional, finalmente es procesado dentro de los `kernels` unidimensionalmente.

Listado 8: Ejemplo de la suma de dos matrices utilizando `CudaMallocPitch`

```
__global__ void kernel(float **gpu_a,size_t pitch1,\
 float **gpu_b,size_t pitch2,float **gpu_c,size_t pitch3){
30
 int idx=threadIdx.x+blockIdx.x*blockDim.x;
 int idy=threadIdx.y+blockIdx.y*blockDim.y;
 float*rowa=(float*)((char*)gpu_a+idx*pitch1);
 float*rowb=(float*)((char*)gpu_b+idx*pitch2);
35 float*rowc=(float*)((char*)gpu_c+idx*pitch3);
 rowc[idy] = rowa[idy] + rowb[idy];
 __syncthreads();
}

40 int main(int argc, char *argv[]) {
 int i;
 int j;
 float a[SIZE][SIZE];float **g_a;
 float b[SIZE][SIZE];float **g_b;
45 float c[SIZE][SIZE];float **g_c;
 dim3 block_size, n_blocks;
 block_size.x = 3;
 block_size.y = 3;
 n_blocks.x = 3;
50 n_blocks.y = 3;

 const int N = SIZE * SIZE;
 for (i = 0; i < SIZE; i++) {
  for (j = 0; j < SIZE; j++) {
```

```
    a[i][j] = sqrt(i+j) ;
    b[i][j] = i+j;
   }
  }
  size_t pitch1;
  cudaMallocPitch(&g_a,&pitch1,SIZE*sizeof(float)
   ,SIZE);
  size_t pitch2;
  cudaMallocPitch(&g_b,&pitch2,SIZE*sizeof(float)
   ,SIZE);
  size_t pitch3;
  cudaMallocPitch(&g_c,&pitch3,SIZE*sizeof(float)
   ,SIZE);

  cudaMemcpy2D(g_a,pitch1,a,SIZE*sizeof(float),
   SIZE*sizeof(float),SIZE,cudaMemcpyHostToDevice);
  cudaMemcpy2D(g_b,pitch2,b,SIZE*sizeof(float),
   SIZE*sizeof(float),SIZE,cudaMemcpyHostToDevice);

  kernel<<<n_blocks,block_size>>>(g_a,pitch1,
   g_b,pitch2,g_c,pitch3);

  cudaMemcpy2D(c,SIZE*sizeof(float),g_c,
   pitch3,SIZE * sizeof(float),SIZE,
   cudaMemcpyDeviceToHost);
```

En en Listado 9, se muestra en mismo ejemplo de la suma de matrices pero utilizando la estructura de datos propuesta en este trabajo. Se puede notar que el código es más legible y mucho mas parecido a estándar `C`.

Listado 9: Ejemplo de la suma de dos matrices utilizando la estructura de datos propuesta

```
void MATRIX_GPU(float ** P,float *** M,\
 int n,int m, char const * var);

float**MATRIX_CPU(int n,int m,char const* var);
21



__global__ void kernel(float**C,float**B,\
 float**A,int ni,int nj){
 int i,j;
 i = blockIdx.x*blockDim.x+threadIdx.x;
 j = blockIdx.y*blockDim.y+threadIdx.y;
 C[i][j] = A[i][j]+B[i][j];
}
```

```
int main(){
 float **h_C=NULL,**dev_C=NULL,*P_C=NULL;
 float **h_A=NULL,**dev_A=NULL,*P_A=NULL;
 float **h_B=NULL,**dev_B=NULL,*P_B=NULL;

 int ni,nj;
 int i,j;
 dim3 grid,block;

 ni = 9;
 nj = 9;

 block.x = 3;
 block.y = 3;
 grid.x = 3;
 grid.y = 3;

 MATRIX_GPU(&P_C, &dev_C, ni, nj, "C");
 MATRIX_GPU(&P_B, &dev_B, ni, nj, "B");
 MATRIX_GPU(&P_A, &dev_A, ni, nj, "A");

 h_C = MATRIX_CPU(ni, nj, "h_C");
 h_B = MATRIX_CPU(ni, nj, "h_B");
 h_A = MATRIX_CPU(ni, nj, "h_A");

 for(i=0;i<ni;i++)
  for(j=0;j<nj;j++){
   h_A[i][j] = sqrt(i+j);
   h_B[i][j] = i + j;
  }

 cudaMemcpy(P_A, h_A[0], ni*nj*sizeof\
  (float),cudaMemcpyHostToDevice);
 cudaMemcpy(P_B, h_B[0], ni*nj*sizeof\
  (float),cudaMemcpyHostToDevice);

 kernel<<<grid,block>>>(dev_C,dev_B,dev_A,ni,nj);
 cudaDeviceSynchronize();
```

En la Figura 4.13 se muestra la estructura del `grid` y de los `bloques` utilizados para la suma de matrices.

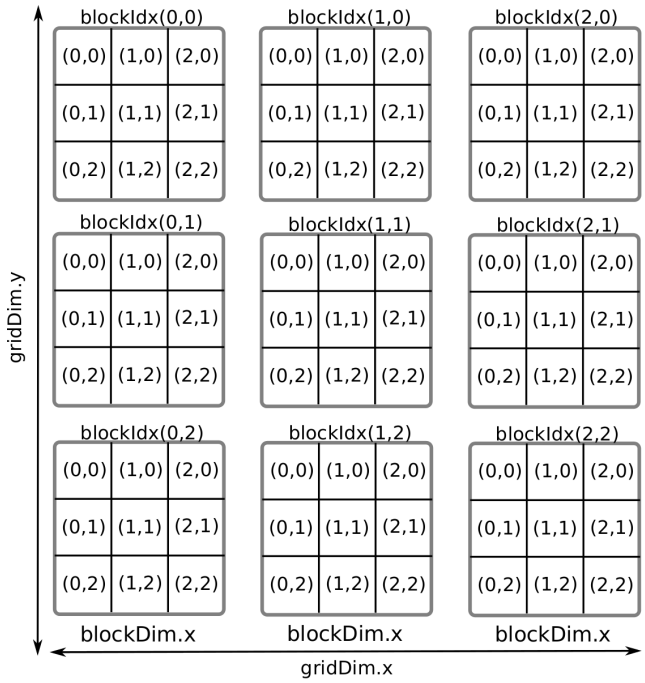


Figura 4.13: Malla utilizada para la suma de matrices.

Para mostrar la facilidad en el uso de la estructura de datos propuesta en un algoritmo más complejo, se resolverá la ecuación de transferencia de calor no estable diferencial parcial en dos y tres dimensiones con variación en el tiempo.

4.3.2 *Solución a la ecuación de transferencia de calor 2D no estacionaria.*

La ecuación bidimensional de transporte de calor no estacionaria,

$$\frac{\partial T}{\partial t} = C\left(\frac{\partial T^2}{\partial x^2} + \frac{\partial T^2}{\partial y^2}\right), \qquad 0 \leqslant x, y \leqslant 1, t \geqslant 0 \tag{9}$$

puede ser escrita para simplificar la notación como:

$$T_t = C(T_{xx} + T_{yy}).$$

los parámetros iniciales :

$$\begin{aligned} T(0, x, y) &= f(x, y), \\ T(t, 0, y) &= \alpha_0(y), \\ T(t, 1, y) &= \alpha_1(y), \\ T(t, x, 0) &= \beta_0(x), \\ T(t, x, 1) &= \beta_1(x). \end{aligned} \tag{10}$$

La solución discreta de la ecuación (9), la llevamos a cabo por medio de diferencias finitas, reemplazando las derivadas continuas por discretas en un punto dado (l, i, j). Al tratarse de un esquema de solución explícito la parcial del tiempo se reemplaza por una diferencia hacia adelante, y se asume que:

$$T_t \approx \frac{T_{i,j}^{l+1} - T_{i,j}^{l}}{\Delta t}, \tag{11}$$

y las derivadas espaciales son reemplazadas por una diferencia centrada de segundo orden,

$$T_{xx} \approx \frac{T_{i+1,j}^{l} - 2T_{i,j}^{l} + T_{i-1,j}^{l}}{\Delta x^2}, \tag{12}$$

$$T_{yy} \approx \frac{T_{i,j+1}^{l} - 2T_{i,j}^{l} + T_{i,j-1}^{l}}{\Delta y^2}, \tag{13}$$

por lo tanto la ecuación (9) es reemplazada por su forma discreta en diferencias como:

$$\frac{T_{i,j}^{l+1} - T_{i,j}^{l}}{\Delta t} = C\left(\frac{T_{i+1,j}^{l} - 2T_{i,j}^{l} + T_{i-1,j}^{l}}{\Delta x^2} + \frac{T_{i,j+1}^{l} - 2T_{i,j}^{l} + T_{i,j-1}^{l}}{\Delta y^2}\right). \tag{14}$$

Reacomodamos los términos para obtener la temperatura de forma explícita en el tiempo $l+1$, es decir en $T_{i,j}^{l+1}$

$$T_{i,j}^{l+1} = T_{i,j}^{l} + C\Delta t\left(\frac{T_{i+1,j}^{l} - 2T_{i,j}^{l} + T_{i-1,j}^{l}}{\Delta x^2} + \frac{T_{i,j+1}^{l} - 2T_{i,j}^{l} + T_{i,j-1}^{l}}{\Delta y^2}\right). \tag{15}$$

Debido a que es una formulación explícita en tiempo, requiere de la siguiente condición de estabilidad:

$$\Delta t \leqslant \frac{1}{2C}\frac{(\Delta x \Delta y)^2}{(\Delta x)^2 + (\Delta y)^2} \tag{16}$$

El dominio computacional es un cuadrado que representa una placa unitaria (1×1), mallada con $\Delta x = 0.01$ y $\Delta y = 0.01$, por lo tanto, $n_x = 101$ y $n_y = 101$, por lo tanto de la (10) las condiciones iniciales impuestas son:

$$f(x, y) = \begin{cases} 1: & \text{Si } 0.05 \leqslant (x-0.5)^2 + (y-0.5)^2 \leqslant 0.1 \\ 0: & \text{en otro caso} \end{cases} \tag{17}$$

y con condiciones a la frontera de tipo Dirichlet

$$\alpha_0(y) = \alpha_1(y) = \beta_0(x) = \beta_1(x) = 0.0 \tag{18}$$

El diagrama de flujo del esquema de la solución se puede ver en la Figura 4.14 y el código fuente en el Listado 17 del Apéndice A.

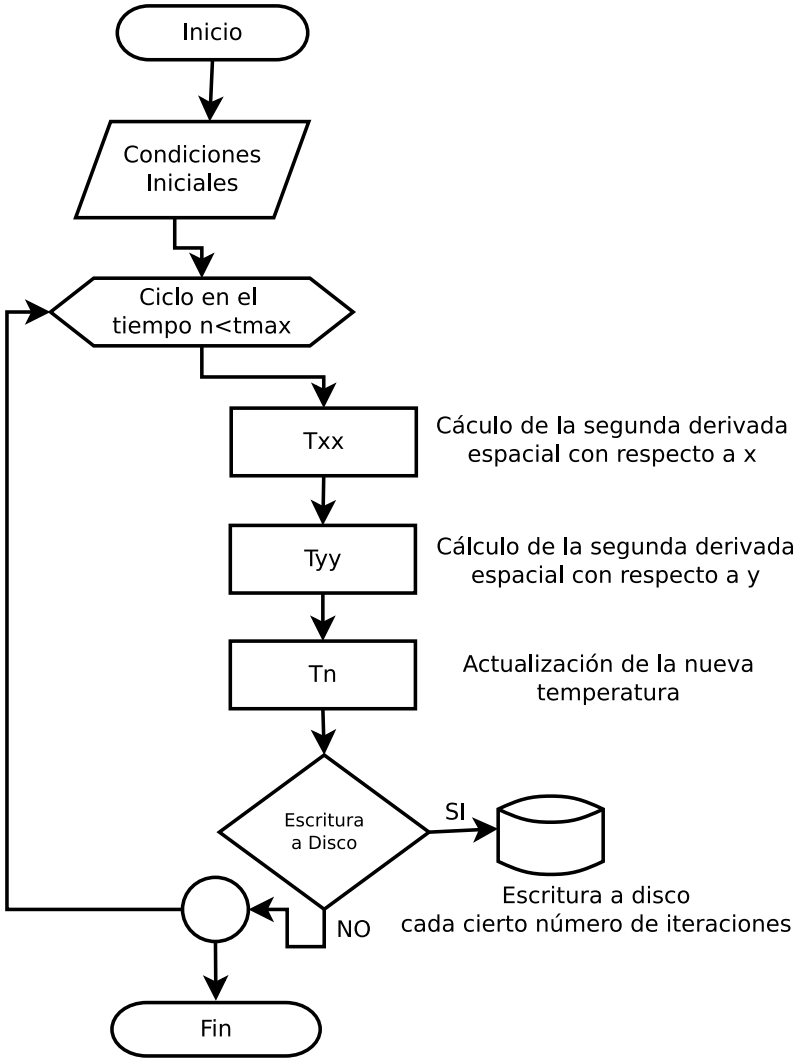


Figura 4.14: Diagrama de flujo correspondiente al código serial para resolver el caso 2D de la ecuación de calor.

Para resolver la ecuación (15) mostramos *tres* diferentes versiones del kernel, que se ejecuta en paralelo para cada punto discreto del plato, de hecho tenemos 201×201 puntos interiores, que son evaluados en paralelo.

En el diagrama de flujo mostrado en la Figura 4.15 se muestra el esquema de ejecución del programa en CUDA, independientemente del tipo de `kernel` que se use.

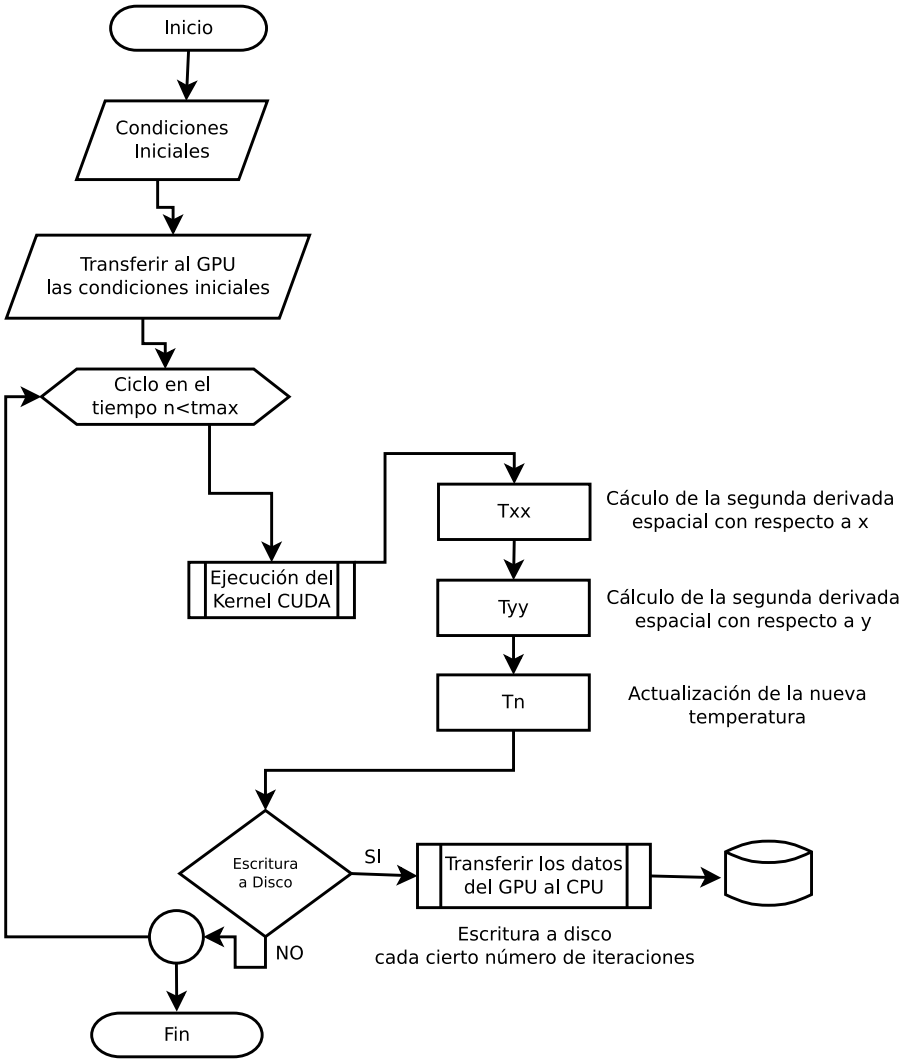


Figura 4.15: Diagrama de flujo correspondiente a la estructura CUDA para resolver la ecuación de calor en 2D.

La primera versión consiste en un kernel con un solo índice, por lo tanto, se debe tratar la malla como un solo renglón, es decir, *renglonizar* [1] la matriz. El código del kernel se muestra en el Listado 10.

1 renglonizar no existe en la gramática española es una anglicismo, que se refiere a convertir una matriz en un vector.

Listado 10: Kernel 2D utilizando un arreglo unidimensional y un solo índice l.

```
__global__ void kernel(float *T, float *Tn, int nx, int ny,\
 float dt, float C, float dx2, float dy2)
{
 int l;
 l = blockIdx.x * blockDim.x + threadIdx.x;
 float Txx;
 float Tyy;
 if(l>nx && l<((nx*ny-1)-nx) && !((l\%nx)==0 || ((l+1)\%nx \
 == 0)))
 {
  Txx = (T[l+1] -2.0f*T[l] + T[l-1])/dx2;
  Tyy = (T[l+nx] -2.0f*T[l] + T[l-nx])/dy2;
  Tn[l] = T[l]+dt*C*(Txx+Tyy);
 }
}
```

En este caso la temperatura T es almacenada en un arreglo bidimensional, accedida por un solo índice l, y el mapeo es definido como $l \rightarrow i \times n_x + j$, donde (i, j) es un punto de la malla de tamaño $n_x \times n_y$. Como $0 \leqslant i \leqslant n_y - 1$ y $0 \leqslant j \leqslant n_x - 1$. Los elementos adyacentes al elemento l del arreglo T están dados por: izquierda $T[l-1]$, derecha $T[l+1]$, arriba $T[l-n_y]$ y abajo $T[l+n_y]$. Los elementos a la frontera pueden ser determinados con las siguientes condicionales:

$$\begin{aligned} &\text{Si} \quad l \bmod n_y = 0 && \text{frontera izquierda} \\ &\text{Si} \quad (l-1) \bmod n_y = 0 && \text{frontera derecha} \\ &\text{Si} \quad l = n_y && \text{frontera superior} \\ &\text{Si} \quad l = n_x n_y - 1 && \text{frontera inferior} \end{aligned} \tag{19}$$

La segunda versión del kernel consiste en crear un arreglo unidimensional pero con dos índices y el código del kernel se muestra en el Listado 11.

En este caso la temperatura T sigue siendo almacenada como un arreglo unidimensional y manejada por dos índices (i, j). Nuevamente $0 \leqslant i \leqslant n_y - 1$ y $0 \leqslant j \leqslant n_x - 1$. Los elementos adyacentes a $T[l]$ son: izquierda $T[i * n_x + j - 1]$, derecha $T[i * n_x + j + 1]$, arriba $T(j + n_x(i-1))$ y abajo $T[j + n_x(i+1)]$. Los elementos a la frontera puede ser determinados con las siguientes condicionales:

$$\begin{aligned} &\text{Si} \quad (in_x + j) \bmod n_x = 0 && \text{frontera izquierda} \\ &\text{Si} \quad (in_x + j) \bmod n_x = 0 && \text{frontera derecha} \\ &\text{Si} \quad (in_x + j) = n_x && \text{frontera superior} \\ &\text{if} \quad (in_x + j) = n_x n_y - 1 && \text{frontera inferior} \end{aligned} \tag{20}$$

Listado 11: Kernel 2D utilizando un arreglo unidimensional y dos índices `(i,j)` .

```
__global__ void kernel(float *T, float *Tn, int ny, int \
 nx, float dt, float C,  float dx2, float dy2)
{
 int i,j,l;
 j = blockIdx.x*blockDim.x+threadIdx.x;
 i = blockIdx.y*blockDim.y+threadIdx.y;
 float Txx;
 float Tyy;
 if( (j > 0) && (j<(nx-1)) && (i > 0) &&  (i < (ny-1)))
 {
  l = i*nx+j;
  Txx = (T[l+1] -2.0f*T[l] + T[l-1])/dx2;
  Tyy = (T[l+nx] -2.0f*T[l] + T[l-nx])/dy2;
  Tn[l] = T[l]+dt*C*(Txx+Tyy);
 }
}
```

El código que contiene el kernel que permite manejar la Temperatura como un arreglo 2D, gracias a la estructura propuesta en este trabajo se muestra en el Listado 12.

Para este caso se almacena la temperatura T como una función del espacio tiempo en un arreglo bidimensional, utilizando la estructura de datos propuesta en este trabajo y manejada de forma simple como en estándar C, con dos índices (i,j). Los elementos adyacentes a $T[i,j]$ ahora triviales de calcular y son: izquierda $T[i,j-1]$, derecha $T[i,j+1]$, arriba $T[i-1,j]$ y abajo $T[i+1,j]$, por lo que los elementos a la frontera pueden ser determinados trivialmente como:

$$\begin{array}{ll} \text{Si } j = 0 & \text{cara izquierda} \\ \text{Si } j = n_x - 1 & \text{cara derecha} \\ \text{Si } \ i = 0 & \text{cara superior} \\ \text{Si } \ i = n_y - 1 & \text{cara inferior} \end{array} \tag{21}$$

Listado 12: Kernel 2D utilizando un arreglo bidimensional y dos índices `(i,j)`.

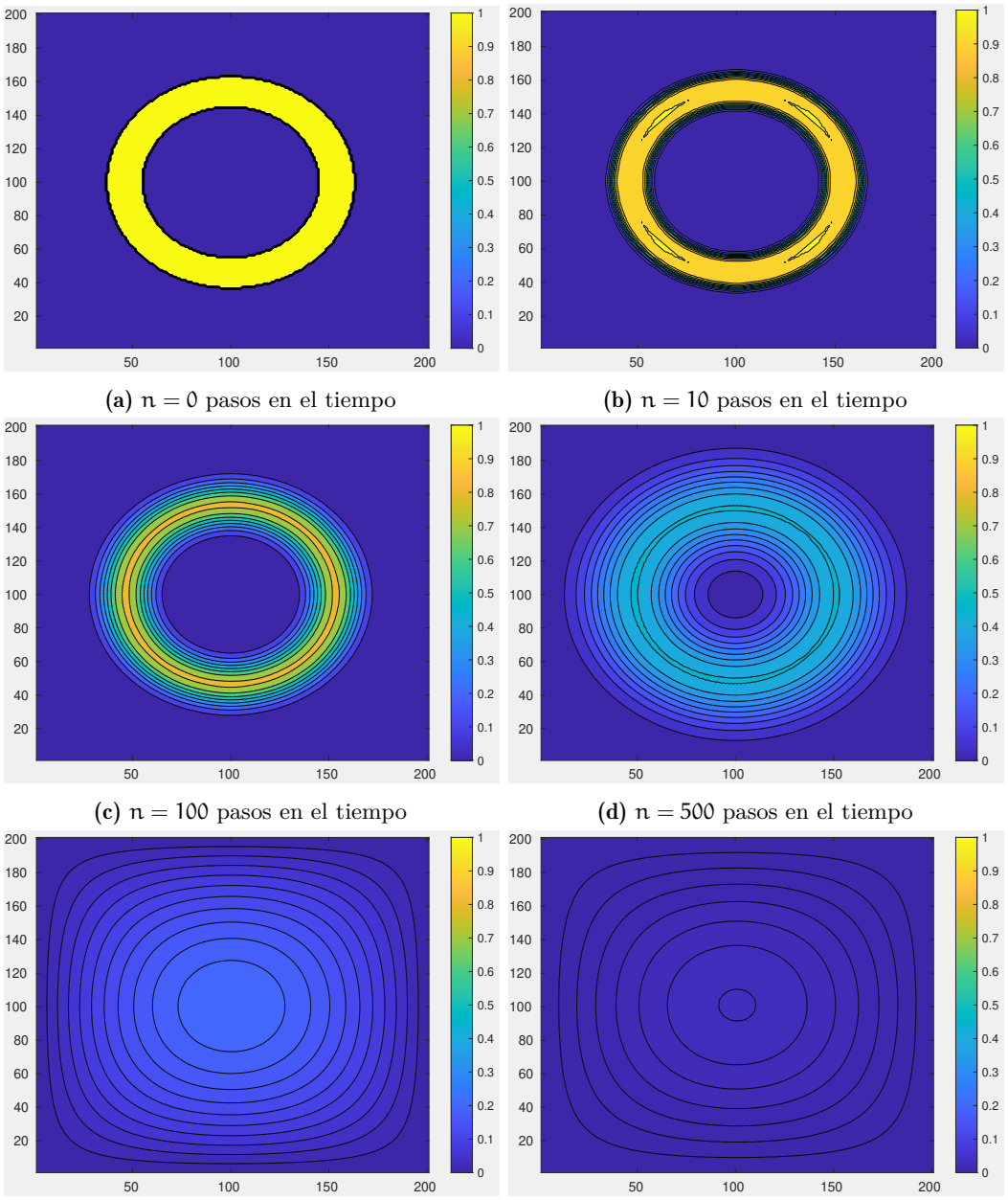
```
__global__ void kernel(float **T, float **Tn, int ny, int\
 nx, float dt, float C, float dx2, float dy2)
{
 int i,j;
 j = blockIdx.x*blockDim.x+threadIdx.x;
 i = blockIdx.y*blockDim.y+threadIdx.y;
```

```
 float Txx;
 float Tyy;
 if( (i > 0) && (i<(ny-1)) && (j > 0) && (j < (nx-1)))
 {
  Txx = (T[i][j+1] -2.0*T[i][j] +T[i][j-1])/dx2;
  Tyy = (T[i+1][j] -2.0*T[i][j] +T[i-1][j])/dy2;
  Tn[i][j] = T[i][j]+dt*C*(Txx+Tyy);
 }
}
```

Para asegurar que cada versión de los `kernels` mostrados en los Listados 10, 11 y 12 están trabajando adecuadamente, se compararon contra la versión serial en estándar `C`, y en cada uno de los tres casos los resultados numéricos son esencialmente los mismos, el resultado de la difusión se muestra en la Figura 4.16 en la cual se muestran los *snapshots*[2] en 0, 10, 100, 500, 5000 y 20000 pasos en el tiempo, en una malla de 201×201 puntos, donde se observa que la difusión es simétrica y la placa se va enfriando.

Los errores absolutos en precisión sencilla (8 dígitos significativos después del punto decimal) fueron calculados para el snapshot en 20,000 pasos en el tiempo, se toma como solución de referencia el programa en C serial y en la Figura 4.18 se muestran los errores obtenidos con cada uno de los `kernels` mostrados en los Listados 10, 11 y 12, los errores rondan alrededor de -6×10^{-6} a 5×10^{-6}, lo que da certeza a la correcta codificación de los `kernels`. Para el cálculo de los errores la malla utilizada es de 301×201 ya que cuando $\mathtt{nx} = \mathtt{ny}$ se pueden disimular errores.

2 En informática, una copia instantánea de volumen o snapshot es una instantánea del estado de un sistema en un momento determinado. El término fue acuñado como una analogía de una fotografía.



(a) $n = 0$ pasos en el tiempo

(b) $n = 10$ pasos en el tiempo

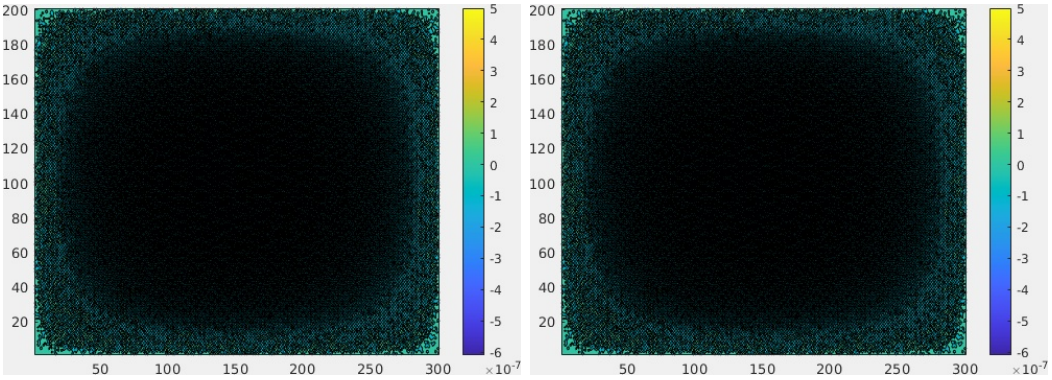
(c) $n = 100$ pasos en el tiempo

(d) $n = 500$ pasos en el tiempo

(e) $n = 5000$ pasos en el tiempo

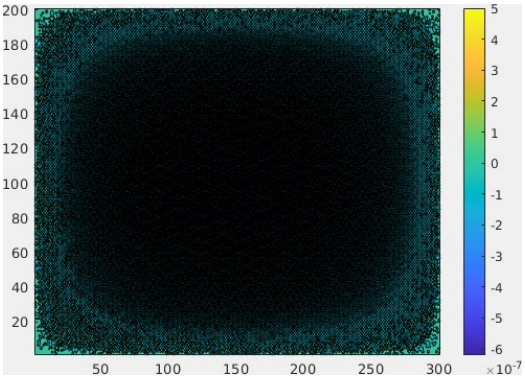
(f) $n = 20000$ pasos en el tiempo

Figura 4.16: Difusión de calor no estacionaria en 2D sobre la placa unitaria generada con la versión en lenguaje C (solución de referencia), las condiciones iniciales se muestran cuando $n = 0$. La malla esta conformada por 201×201 puntos, por lo tanto, $\Delta x = 0.005$ y $\Delta y = 0.005$, Δt se calcula con el criterio CFL.



(a) Resultado obtenido con el kernel del Listado 10.

(b) Resultado obtenido con el kernel del Listado 11.



(c) Resultado obtenido con el kernel del Listado 12.

Figura 4.17: Error absoluto con respecto a la solución de referencia después de 20,000 pasos en el tiempo ($n = 20,000$), el error máximo encontrado es del orden de $10^{-}7$, lo cual es aceptable para la precisión de tipo `float`.

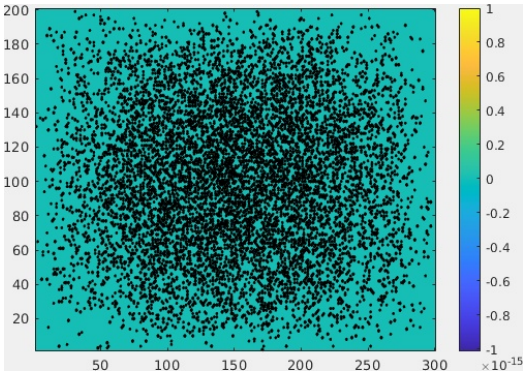
4.3.3 *Solución a la ecuación de transferencia de calor 3D no estacionaria.*

La ecuación tridimensional de transporte de calor no estacionaria,

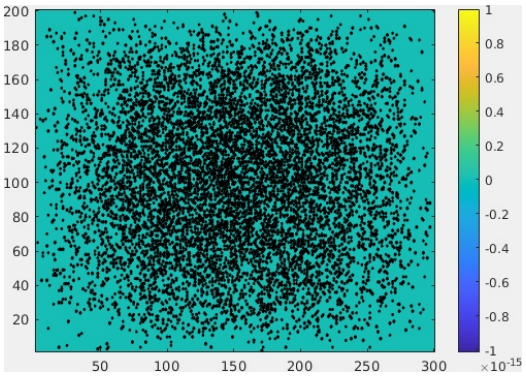
$$\frac{\partial T}{\partial t} = C\left(\frac{\partial T^2}{\partial x^2} + \frac{\partial T^2}{\partial y^2} + \frac{\partial T^2}{\partial z^2}\right) \qquad 0 \leqslant x, y, z \leqslant 1, t \geqslant 0 \tag{22}$$

simplificando la notación puede ser reescrita como:

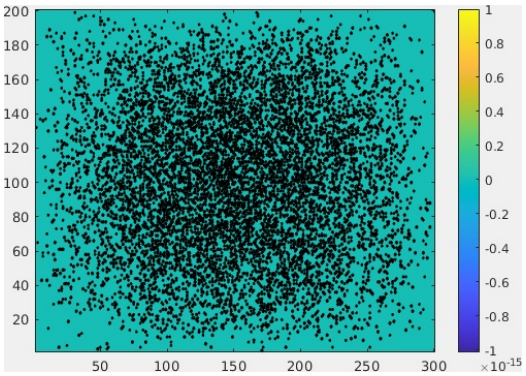
$$T_t = C(T_{xx} + T_{yy} + T_{zz}) \tag{23}$$



(a) Resultado obtenido con el kernel del Listado 10.



(b) Resultado obtenido con el kernel del Listado 11.



(c) Resultado obtenido con el kernel del Listado 12.

Figura 4.18: Error absoluto con respecto a la solución de referencia después de 20,000 pasos en el tiempo ($n = 20,000$), el error máximo encontrado es del orden de $10^{-}15$, lo cual es aceptable para la precisión de tipo `double`.

Las condiciones iniciales y de frontera son:

$$\begin{aligned}
T(0,x,y,z) &= f(x,y,z),\\
T(t,0,y,z) &= \alpha_0(y,z),\\
T(t,1,y,z) &= \alpha_1(y,z),\\
T(t,x,0,z) &= \beta_0(x,z),\\
T(t,x,1,z) &= \beta_1(x,z),\\
T(t,x,y,0) &= \gamma_0(x,y),\\
T(t,x,y,1) &= \gamma_1(x,y).
\end{aligned} \tag{24}$$

Para aproximar una solución discreta, hay que reemplazar las derivadas continuas por una expresión algebraica en diferencias finitas. Específicamente, en un punto dado (l,i,j,z), se reemplazará la parcial del tiempo por una diferencia hacia adelante

con la finalidad de dejar la formulación explícita en tiempo y las derivadas espaciales por una diferencia centrada de segundo orden como:

$$T_t \approx \frac{T^{l+1}_{i,j,k} - T^{l}_{i,j,k}}{\Delta t}, \tag{25}$$

$$T_{xx} \approx \frac{T^{l}_{i+1,j,k} - 2T^{l}_{i,j,k} + T^{l}_{i-1,j,k}}{\Delta x^2}, \tag{26}$$

$$T_{yy} \approx \frac{T^{l}_{i,j+1,k} - 2T^{l}_{i,j,k} + T^{l}_{i,j-1,k}}{\Delta y^2}, \tag{27}$$

$$T_{zz} \approx \frac{T^{l}_{i,j,k+1} - 2T^{l}_{i,j,k} + T^{l}_{i,j,k-1}}{\Delta y^2}. \tag{28}$$

Por lo tanto la ecuación (22), es reemplazada por su aproximación algebraica en diferencias finitas como:

$$\begin{aligned}\frac{T^{l+1}_{i,j,k} - T^{l}_{i,j,k}}{\Delta t} = C \Bigg(& \frac{T^{l}_{i+1,j,k} - 2T^{l}_{i,j,k} + T^{l}_{i-1,j,k}}{\Delta x^2} \\ & + \frac{T^{l}_{i,j+1,k} - 2T^{l}_{i,j,k} + T^{l}_{i,j-1,k}}{\Delta y^2} + \frac{T^{l}_{i,j,k+1} - 2T^{l}_{i,j,k} + T^{l}_{i,j,k-1}}{\Delta z^2} \Bigg).\end{aligned} \tag{29}$$

Reacomodando los términos para obtener la temperatura explícita en el tiempo $l+1$ queda como:

$$\begin{aligned}T^{l+1}_{i,j,k} = T^{l}_{i,j,k} + C\Delta t \Bigg(& \frac{T^{l}_{i+1,j,k} - 2T^{l}_{i,j,k} + T^{l}_{i-1,j,k}}{\Delta x^2} \\ & + \frac{T^{l}_{i,j+1,k} - 2T^{l}_{i,j,k} + T^{l}_{i,j-1,k}}{\Delta y^2} + \frac{T^{l}_{i,j,k+1} - 2T^{l}_{i,j,k} + T^{l}_{i,j,k-1}}{\Delta z^2} \Bigg).\end{aligned} \tag{30}$$

Debido a que es un esquema explícito, requiere la siguiente condición de estabilidad para el avance en el tiempo:

$$\Delta t \leqslant \frac{1}{2C} \frac{(\Delta x \Delta y \Delta z)^2}{(\Delta x)^2(\Delta y)^2 + (\Delta x)^2(\Delta z)^2 + (\Delta y)^2(\Delta z)^2} \tag{31}$$

Para este caso el dominio computacional es un cubo de $1 \times 1 \times 1$, y las condiciones iniciales están dadas por:

$$f(x,y,z) = \begin{cases} 1: & \text{si } 0.05 \leqslant (x-0.5)^2 + (y-0.5)^2 + (z-0.5)^2 \leqslant 0.1 \\ 0: & \text{en otro caso.} \end{cases} \tag{32}$$

y las condiciones a la frontera de Dirichlet,

$$\alpha_0(y,z) = \alpha_1(y,z) = \beta_0(x,z) = \beta_1(x,z) = \gamma_0(x,y) = \gamma_1(x,y) = 0.0 \qquad (33)$$

El diagrama de flujo del esquema de la solución para el caso 3D en forma serial, se puede ver en la Figura 4.19 y el código fuente serial 18 en el Apéndice A.

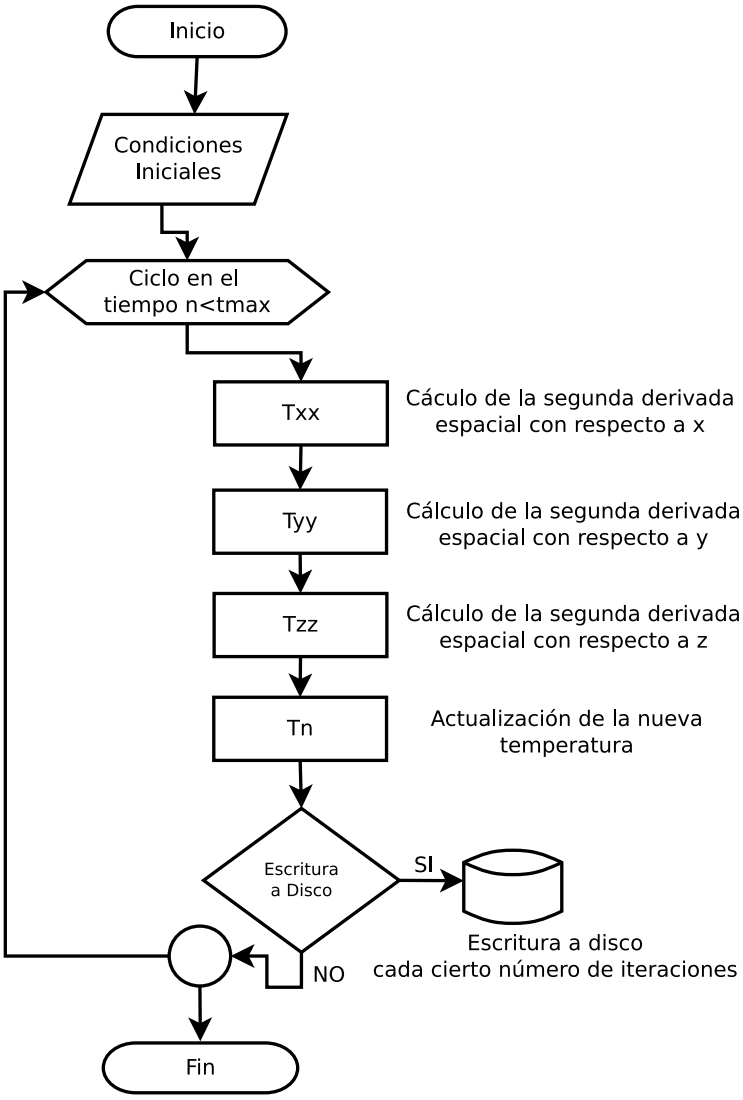


Figura 4.19: .

Para el caso 3D se programaron tres soluciones que implican tres diferentes implementaciones de `kernels`, el primero utilizando un arreglo unidimensional procesado con un índice, un arreglo unidimensional procesado con tres índices, y un arreglo

tridimensional con tres índices, aporte principal de este trabajo. La solución serial en estándar `C` sirve como solución de referencia.

El diagrama de flujo mostrado en la Figura 4.20 se muestra el esquema de ejecución del programa en CUDA, no obstante del `kernel` que se utilice.

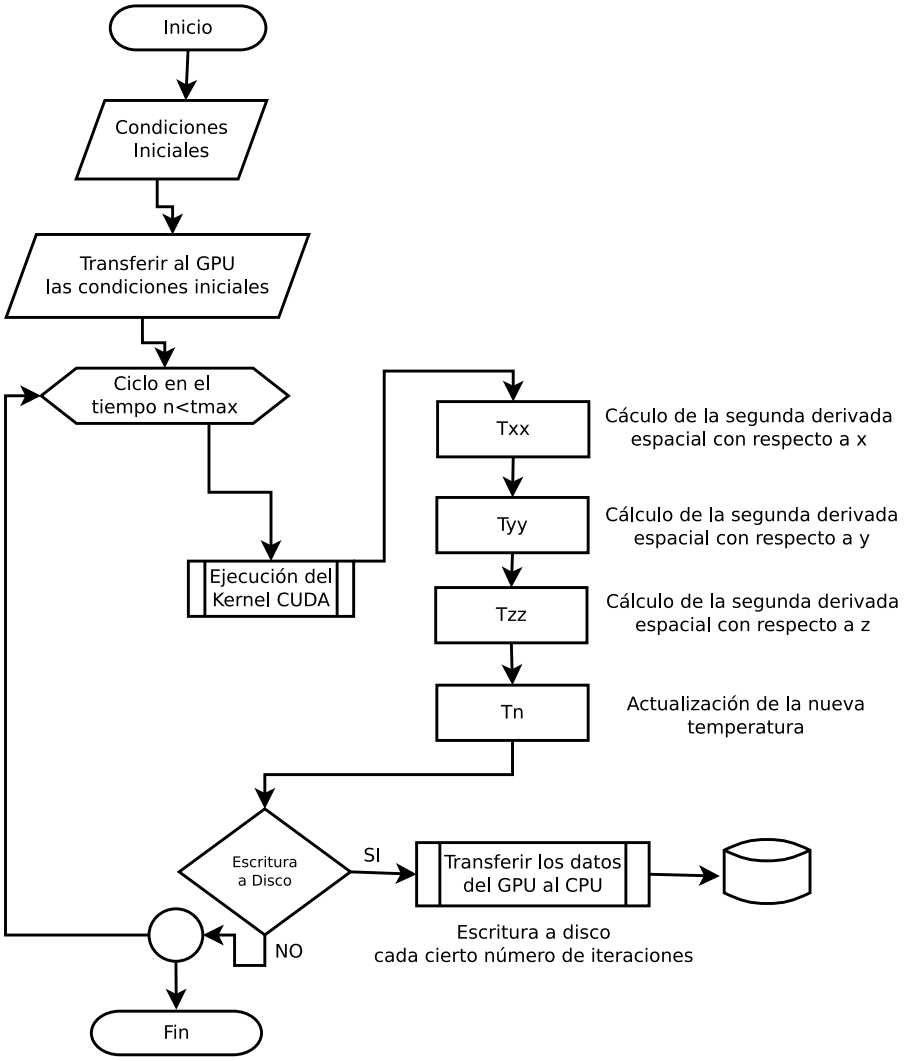


Figura 4.20: Diagrama de flujo correspondiente a la estructura CUDA para resolver la ecuación de calor en 3D.

El primer kernel desarrollado para resolver la ecuación (30), es mostrado en el Listado 13. En este caso la temperatura $\mathtt{T}$ es almacenada en un arreglo unidimensional y manejada por un solo índice $\mathtt{l}$, el mapeo esta dado por $\mathtt{l}(\mathtt{i},\mathtt{j},\mathtt{k}) \rightarrow \mathtt{j} + \mathtt{i}n_x + \mathtt{k}n_x n_y$, donde $\mathtt{i}$, $\mathtt{j}$ y $\mathtt{k}$ es la posición de un punto discreto dentro de la malla de tamaño

$n_x \times n_y \times n_z$. El dominio discreto esta dado por: $0 \leqslant i \leqslant n_x - 1$, $0 \leqslant j \leqslant n_y - 1$ and $0 \leqslant k \leqslant n_z - 1$ y los elementos adyacentes a $T[l]$ son: izquierda $T[l-1]$, derecha $T[l+1]$, arriba $T[l-n_y]$, abajo $T[l+n_y]$, enfrente $T[l+n_y]$ y atrás $T[l+n_x n_y]$. Con esta estructura los puntos en las caras del cubo pueden ser determinadas, ya que los cálculos se llevan a cabo sobre los puntos interiores. Los elementos a la frontera pueden ser determinados con las siguientes condicionales:

$$\begin{array}{ll} \text{Si } \; l \bmod n_x = 0 & \text{cara izquierda} \\ \text{Si } \; (l+1) \bmod n_x = 0 & \text{cara derecha} \\ \text{Si } \; l \bmod (n_x n_y) < n_x & \text{cara superior} \\ \text{Si } \; (l - n_x n_y + n_x) \bmod n_x n_y < n_x & \text{cara inferior} \\ \text{Si } \; l > n_x n_y & \text{cara frontal} \\ \text{Si } \; l < n_x n_y (n_z - 1) & \text{cara trasera} \end{array} \tag{34}$$

Listado 13: Kernel 3D utilizando un arreglo unidimensional y un solo índice `l`. .

```
__global__ void kernel(float *T, float *Tn, int ny, int \
 nx, int nz, float dt, float C, float dx2, float dy2, \ float
     dz2)
{
 int l;
 float Txx;
 float Tyy;
 float Tzz;
 l = blockIdx.x * blockDim.x + threadIdx.x;
 if( l > (nx*ny) && l < ((nx*ny)*(nz-1)) && !((l\%nx)==0 \
 || ((l+1)\%nx == 0)) && !((l\%(nx*ny) < nx) || \
   (((l-(nx*ny)+nx)\%((nx*ny))<nx) ) ))
 {
  Txx = (T[l+1] -2.0f*T[l] +T[l-1])/dx2;
  Tyy = (T[l+nx] -2.0f*T[l] +T[l-nx])/dy2;
  Tzz = (T[l+nx*ny] -2.0f*T[l] +T[l-nx*ny])/dz2;
  Tn[l] = T[l]+dt*C*(Txx+Tyy+Tzz);
 }
}
```

Para el segundo `kernel` mostrado en el Listado 14, la temperatura T es almacenada en un arreglo unidimensional y manejada por tres índices (i, j, k) que representa un elemento dentro de la malla tridimensional $n_x \times n_y \times n_z$. Este `kernel` se muestra en el Listado 14, y nuevamente el dominio discreto esta dado por: $0 \leqslant i \leqslant n_x - 1$, $0 \leqslant j \leqslant n_y - 1$ and $0 \leqslant k \leqslant n_z - 1$ y los elementos adyacentes a $T[l]$, $(l(i, j, k) \rightarrow j + i n_x + k n_x n_y)$, son: izquierda $T[j + i n_x + k n_x n_y - 1]$, derecha

$T[j+in_x+kn_xn_y+1]$, arriba $T[j+n_x(i+kn_y-1)]$, abajo $T[j+n_z(i+kn_y+1)]$, frente $T[j+n_x(i+n_y(k-1))]$ y atrás $T[j+n_x(i+n_y(k+1))]$. Los elementos a la frontera con tres índices pueden ser determinados como:

$$\begin{aligned}
&\text{if } (j+n_x(i+kn_y)) \bmod n_x = 0 && \text{cara izquierda}\\
&\text{if } (j+n_x(i+kn_y)+1) \bmod n_x = 0 && \text{cara derecha}\\
&\text{if } (j+n_x(i+kn_y)) \bmod (n_xn_y) < n_x && \text{cara superior}\\
&\text{if } (j+n_x(i+n_y(k-1)+1)) \bmod n_xn_y < n_x && \text{cara inferior}\\
&\text{if } (j+n_x(i+kn_y)) > n_xn_y && \text{cara frontal}\\
&\text{if } (j+n_x(i+kn_y)) < n_xn_y(n_z-1) && \text{cara trasera}
\end{aligned} \tag{35}$$

Listado 14: Kernel 3D utilizando un arreglo unidimensional y tres índices `(i,j,k)`. .

```
__global__ void kernel(float *T, float *Tn, int ny, int \
 nx, int nz, float dt, float C, float dx2, float dy2, \
 float dz2)
{
 int i,j,k,l;
 j = blockIdx.x * blockDim.x + threadIdx.x;
 i = blockIdx.y * blockDim.y + threadIdx.y;
 k = blockIdx.z * blockDim.z + threadIdx.z;
 float Txx;
 float Tyy;
 float Tzz;
 if( (j>0) && (j<(nx-1)) && (i > 0) \
  && (i < (ny-1)) && (k>0) && (k<(nz-1)) )
 {
  l = j+i*nx+k*(nx*ny);
  Txx = (T[l+1] -2.0f*T[l] +T[l-1])/dx2;
  Tyy = (T[l+nx] -2.0f*T[l] +T[l-nx])/dy2;
  Tzz = (T[l+nx*ny] -2.0f*T[l] +T[l-nx*ny])/dz2;
  Tn[l] = T[l]+dt*C*(Txx+Tyy+Tzz);
 }
}
```

Para el tercer `kernel` la Temperatura es almacenada en un arreglo tridimensional construido con la estructura de datos desarrollada en este trabajo, y manejada por tres índices `(i,j,k)` y el `kernel` se muestra en el Listado 15. Los elementos adyacentes a `T[i,j,k]` son: izquierdo `T[i,j-1,k]`, derecho `T[i,j+1,k]`, arriba `T[i-1,`

`j,k]`, abajo `T[i+1,j,k]`, enfrente `T[i,j,k-1]` y atrás `T[i,j,k+1]`. Los elementos a la frontera pueden ser claramente determinados con las siguientes condicionales:

$$\begin{aligned}
&\text{if } j > 0 && \text{cara izquierda}\\
&\text{if } j < n_x - 1 && \text{cara derecha}\\
&\text{if } i > 0 && \text{cara superior}\\
&\text{if } i < n_y - 1 && \text{cara inferior}\\
&\text{if } k > 0 && \text{cara frontal}\\
&\text{if } k < n_z - 1 && \text{cara trasera}
\end{aligned} \tag{36}$$

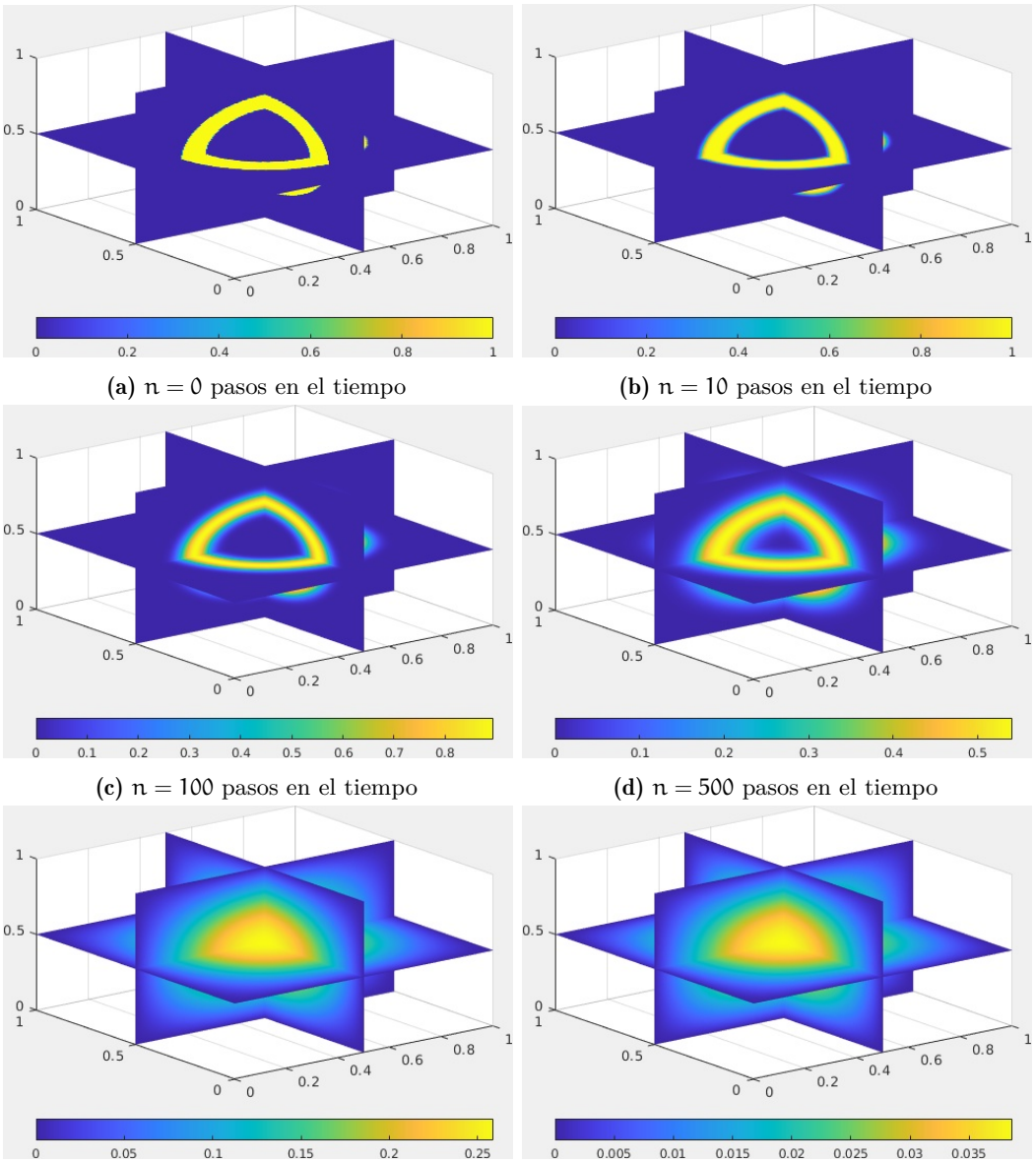
Listado 15: Kernel 3D utilizando un arreglo unidimensional y tres índices `(i,j,k)`. .

```
__global__ void kernel(real ***T, real ***Tn, int ny, int nx,
    int nz, real dt,
real C, real dx2, real dy2, real dz2) {
  int i, j, k;
  j = blockIdx.x * blockDim.x + threadIdx.x;
  i = blockIdx.y * blockDim.y + threadIdx.y;
  k = blockIdx.z * blockDim.z + threadIdx.z;
  real Txx;
  real Tyy;
  real Tzz;
  if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1)) &&
      (k > 0) &&
      (k < (nz - 1))) {
    Tzz = (T[i][j][k + 1] - 2.0 * T[i][j][k] + T[i][j][k - 1])
        / dz2;
    Txx = (T[i][j + 1][k] - 2.0 * T[i][j][k] + T[i][j - 1][k])
        / dx2;
    Tyy = (T[i + 1][j][k] - 2.0 * T[i][j][k] + T[i - 1][j][k])
        / dy2;

    Tn[i][j][k] = T[i][j][k] + dt * C * (Txx + Tyy + Tzz);
  }
}
```

4.3.4 *Consideraciones de rendimiento, el uso de memoria compartida*

Es bien sabido que el uso de memoria compartida en los GPUs puede mejorar notablemente el rendimiento de algunos algoritmos (Winkler et al., 2017; Rosen,



(a) $n = 0$ pasos en el tiempo

(b) $n = 10$ pasos en el tiempo

(c) $n = 100$ pasos en el tiempo

(d) $n = 500$ pasos en el tiempo

(e) $n = 5000$ pasos en el tiempo

(f) $n = 20000$ pasos en el tiempo

Figura 4.21: Difusión de calor no estacionaria en 3D sobre el cubo unitario generado con la versión en lenguaje C (solución de referencia), las condiciones iniciales se muestran cuando $n = 0$. La malla esta conformada por $201 \times 201 \times 201$ puntos, por lo tanto, $\Delta x = 0.005$, $\Delta y = 0.005$ y $\Delta z = 0.005$.

2013). La memoria compartida contenida en cada multiprocesador *stream* ya que es mas rápida que la memoria global, de hecho, la latencia de la memoria global es de alrededor de 100X más lenta.

La memoria compartida es asignada por bloque, por lo tanto todos los hilos del mismo tienen acceso a la memoria compartida, y en un problema de diferencias finitas, el uso de este tipo de memoria puede incrementar el rendimiento debido a que el número de accesos a la memoria global es reducido.

El uso de memoria compartida, puede complicar la codificación del programa, sin embargo, en este trabajo, mostramos una forma clara y relativamente fácil, para conservar la legibilidad y mantenimiento del código.

La idea principal consiste en tener un arreglo por cada bloque en el cual se copia la información correspondiente del arreglo principal, se calculan las diferencias finitas en un arreglo compartido local y el resultado se regresa a la memoria global. Para tal implementación utilizaremos los arreglos compartidos `tile` and `tilen`.

En la Figura 4.22 se muestra el dominio computacional, donde una partición del dominio almacenada `tilen`, tiene una región adicional de dos columnas y dos renglones en la frontera, que son necesarios para llevar a cabo el cálculo de la molécula computacional, es decir, que requiere de las fronteras vecinas para poder completar el cálculo `tilen` es el incremento del dominio con las regiones sombra.

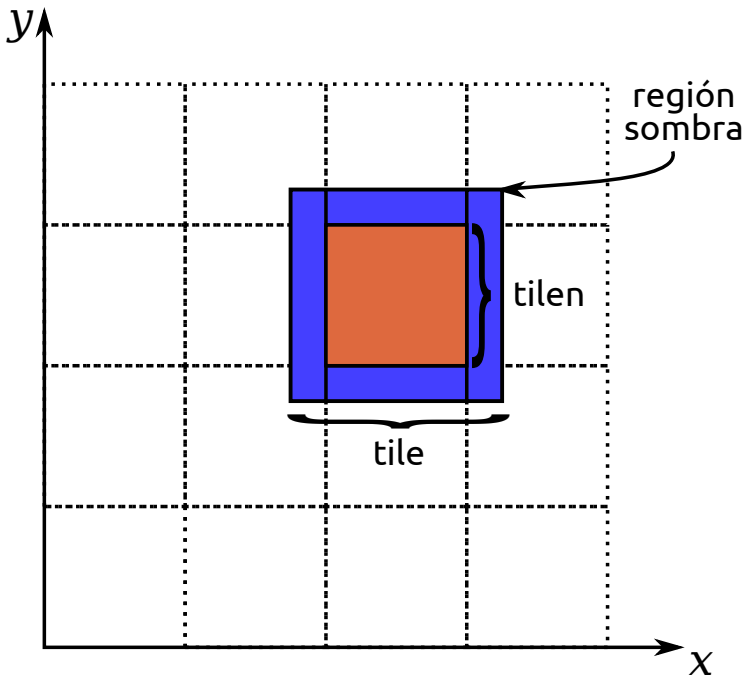


Figura 4.22: Uso de memoria compartida en una malla 2D, la zona azul representa la región aumentada o región sombra que permite completar los cálculos en la parte interior.

El uso de memoria compartida para el código del kernel mostrado en el Listado 12, se muestra en el Listado 16.

Listado 16: kernel 2D para el cálculo de la temperatura T utilizando el arreglo 2D propuesto en este artículo, con dos índices i, j y usando memoria. compartida

```
int i,j,l,m;
float Txx;
float Tyy;

__shared__ float tile[tiley+2][tilex+2];
__shared__ float tilen[tiley][tilex];

j = blockIdx.x*blockDim.x+threadIdx.x;
i = blockIdx.y*blockDim.y+threadIdx.y;

if((i >= 0) && (i <= (ny-1)) && (j >= 0) && (j <= (nx-1)))
  tile[threadIdx.y+1][threadIdx.x+1] = T[i][j];

if((i > 0) && (i < (ny-1)) && (j > 0) && (j < (nx-1))){
  if(threadIdx.x == (blockDim.x-1))
```

```
    tile[threadIdx.y+1][threadIdx.x+2] = T[i][j+1];
  if(threadIdx.y == (blockDim.y-1))
    tile[threadIdx.y+2][threadIdx.x+1] = T[i+1][j];
  if(threadIdx.x == (0))
    tile[threadIdx.y+1][threadIdx.x]   = T[i][j-1];
  if(threadIdx.y == (0))
    tile[threadIdx.y][threadIdx.x+1]   = T[i-1][j];
}
__syncthreads();

if((i > 0) && (i < (ny-1)) && (j > 0) && (j < (nx-1))){
  m = threadIdx.x+1;
  l = threadIdx.y+1;
  Txx = (tile[l][m+1]-2.0f*tile[l][m]+tile[l][m-1])/dx2;
  Tyy = (tile[l+1][m]-2.0f*tile[l][m]+tile[l-1][m])/dy2;
  tilen[l-1][m-1] = tile[l][m]+dt*C*(Txx+Tyy);
}

if(j==0)
  tilen[threadIdx.y][threadIdx.x] = 0.0f;
if(j==(nx-1))
  tilen[threadIdx.y][threadIdx.x] = 0.0f;
if(i==0)
  tilen[threadIdx.y][threadIdx.x] = 0.0f;
if(i==(ny-1))
  tilen[threadIdx.y][threadIdx.x] = 0.0f;

if((i >= 0) && (i <= (ny-1)) && (j >= 0) && (j <= (nx-1)))
  Tn[i][j] = tilen[threadIdx.y][threadIdx.x];
```

En el Listado 16 podemos observar la introducción de los dos nuevos arreglos `tile` y `tilen`. El primero (`tile`) sirve para extraer de la memoria global el subconjunto de la malla correspondiente al bloque y `tilen`

Es muy importante tener en cuenta, que la memoria compartida podría no ser útil, por la siguiente razón: cada elemento es utilizado solo una vez. Para que la memoria compartida muestre una utilidad real y se vea reflejada en el rendimiento de la aplicación, la memoria compartida debe tener varios accesos, lo cual se puede incrementar, utilizando buenos patrones de diseño.

4.3.5 *Pruebas de Rendimiento*

Los experimentos se llevaron a cabo sobre una tarjeta de gama media, la GeForce RTX 2060 Super, con CUDA 10.2 y `gcc` 7.5

Las características principales de la RTX 2060 Super son:

- 34 multi-processors (SM),
- El tamaño del `warp` es de 32 hilos,
- 8 GB of DDR6 global memory,
- Número máximo de hilos por bloque 1024.
- Dimensiones máximas por bloque $1024(x)$, $1024(y)$, $64(z)$.
- Tamaño máximo del grid (malla) $2147483647 \times 65535 \times 65535$

Para entender el significado de las pruebas de rendimiento hay que clarificar los siguientes conceptos:

- El programador escribe un `kernel` y organiza su ejecución en una malla de bloques.
- Cada bloque es asignado a un multiprocesador (SM), y una vez asignado no puede migrar a otro SM.
- Cada SM divide cada uno de sus bloques en `warps`, es decir, en 32 hilos, y estos 32 hilos se ejecutarán concurrentemente, si en el bloque hay más hilos estos esperarán su turno al siguiente `warp`.
- La ejecución de un hilo es llevada a cabo por un núcleo CUDA contenido en un SM, y no existe un mapeo específico entre hilos y núcleos, es decir, no se especifica ninguna afinidad. No obstante, si un hilo detiene su ejecución por acceso a memoria, su ejecución puede pasar a diferente núcleo.
- En presencia de bifurcaciones (`condicionales`) que causen caminos diferentes a seguir por los hilos dentro de un `warp` (ramas divergentes) se serializa la ejecución de cada rama, deshabilitando hilos que no sigan la rama activa.
- Una vez que las ramas divergentes se completan, los hilos convergen nuevamente.

Los tiempos de cómputo obtenidos después de 20,000 pasos en el tiempo, utilizando la RTX 2060 Super utilizando los `kernels` para el problema 2D en precisión sencilla (`float`) son mostrados en milisegundos en la Tabla 4.5 y para precisión doble (`double`) en la Tabla 4.6. Las configuraciones de los bloques fueron seleccionadas en base al tamaño del `warp`: ½`warp`, un `warp`, dos `warp`, cuatro `warp`, ocho `warp`, hasta llegar el máximo número de hilos que es 32 `warp` (1024 hilos). Para el caso particular del `kernel` 1D, los bloques solo tienen en tamaño en una sola dimensión.

El tamaño del grid (malla) es calculado automáticamente en base al tamaño del bloque y del dominio. Las notaciones son 1D-1, un arreglo unidimensional utilizando un índice; 1D-2, un arreglo unidimensional usando 2 índices; 2D-2 un arreglo bidimensional utilizando 2 índices y la letra S índica el uso de memoria compartida.

Tabla 4.5: Tiempos de cómputo en milisegundos obtenidos después de 20000 pasos en el tiempo en precisión sencilla (`float`), se incluye el tiempo necesario para escribir el resultado final el cual es el mismo para todos los casos.

Block Size	1D-1	1D-2	1D-2-S	2D-2	2D-2-S
4×4 (16)	211	207	233	629	239
8×4 (32)	151	138	155	379	183
8×8 (64)	103	106	123	373	141
16×8 (128)	98	93	117	373	137
16×16 (256)	104	98	117	370	140
32×16 (512)	100	93	112	390	130
32×32 (1024)	104	104	133	402	151

Tabla 4.6: Tiempos de cómputo en milisegundos obtenidos después de 20000 pasos en el tiempo en precisión doble (`double`)), se incluye el tiempo necesario para escribir el resultado final el cual es el mismo para todos los casos.

Block Size	1D-1	1D-2	1D-2-S	2D-2	2D-2-S
4×4 (16)	529	523	525	525	538
8×4 (32)	328	332	316	302	338
8×8 (64)	313	301	349	303	318
16×8 (128)	315	297	326	298	358
16×16 (256)	309	335	324	326	343
32×16 (512)	361	326	338	333	361
32×32 (1024)	375	359	396	367	397

Los tiempos de cómputo obtenidos para el caso de la aplicación 3D, con 20,000 pasos en el tiempo, son mostrados en las Tablas 4.7 y 4.8 respectivamente.

Tabla 4.7: Tiempos de cómputo en milisegundos obtenidos después de 20000 pasos en el tiempo en precisión sencilla (`float`) para el caso 3D, se incluye el tiempo necesario para escribir el resultado final el cual es el mismo para todos los casos.

Block Size	1D-1	1D-3	1D-3-S	3D-3	3D-3-S
$4 \times 4 \times 4$ (64)	59179	59795	59474	59522	56542
$8 \times 4 \times 4$ (128)	59503	59483	58841	60068	56952
$8 \times 8 \times 4$ (256)	59114	58970	59440	59638	57011
$8 \times 8 \times 8$ (512)	59231	59004	58718	59867	49993
$16 \times 8 \times 8$ (1024)	62520	61641	62631	65925	68582

Tabla 4.8: Tiempos de cómputo en milisegundos obtenidos después de 20000 pasos en el tiempo en precisión doble (`double`)), se incluye el tiempo necesario para escribir el resultado final el cual es el mismo para todos los casos.

Block Size	1D-1	1D-3	1D-3-S	3D-3	3D-3-S
$4 \times 4 \times 4$ (64)	46708	46642	46373	53704	56655
$8 \times 4 \times 4$ (128)	46716	46233	46760	54128	56934
$8 \times 8 \times 4$ (256)	45873	46558	46878	54278	56782
$8 \times 8 \times 8$ (512)	46942	46191	46526	47606	50203
$16 \times 8 \times 8$ (1024)	49723	49585	49955	56311	69200

Los tiempos de cómputo contenidos en la Tabla 4.23 son graficados en la Figura 4.23, donde se observa que la versión más común para la implementación en diferencias finitas (1D-2) y (1D-1) son apenas perceptiblemente más rápidas, y la versión propuesta (2D-2) es más lenta, de hecho se ve su separación lejos del comportamiento de las demás curvas, sin embargo, aquí entra en juego la memoria compartida, en las configuración (1D-2-S) la memoria compartida no tiene beneficio, sin embargo, para la estructura propuesta (2D-2-S) el uso de la memoria compartida tiene una mejora significativa acercándose al tiempo estimado en la familia de curvas (1D-1,1D-2,1D-2-S). No obstante, este fenómeno con la memoria compartida no sucede en precisión doble (`double`), como puede observarse en la Figura 4.24, aunque la configuración (1D-2) es ligeramente más rápida que todas, realmente no existe diferencia significativa con la estructura propuesta (2D-2),(2D-2-S), y la memoria compartida no hace ninguna diferencia.

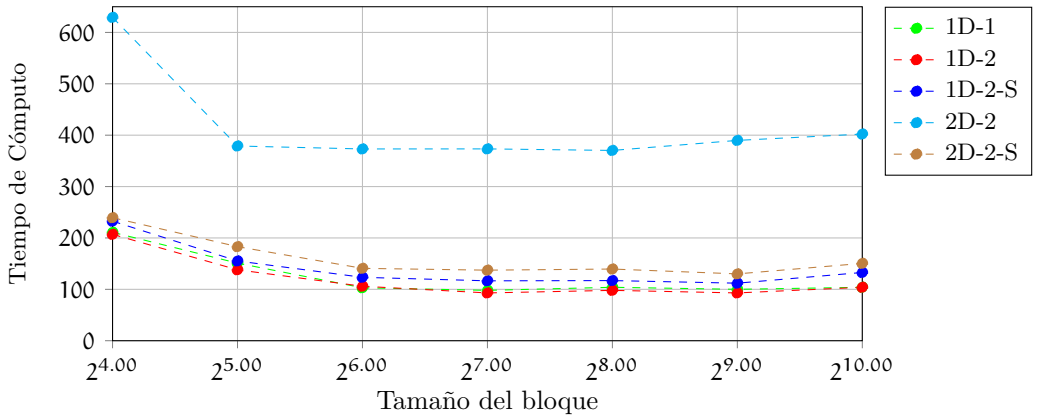


Figura 4.23: Comportamiento del tiempo de cómputo en milisegundos, obtenidos probando diferentes configuraciones de los kernels 2D en precisión sencilla mostrados en la Tabla 4.5.

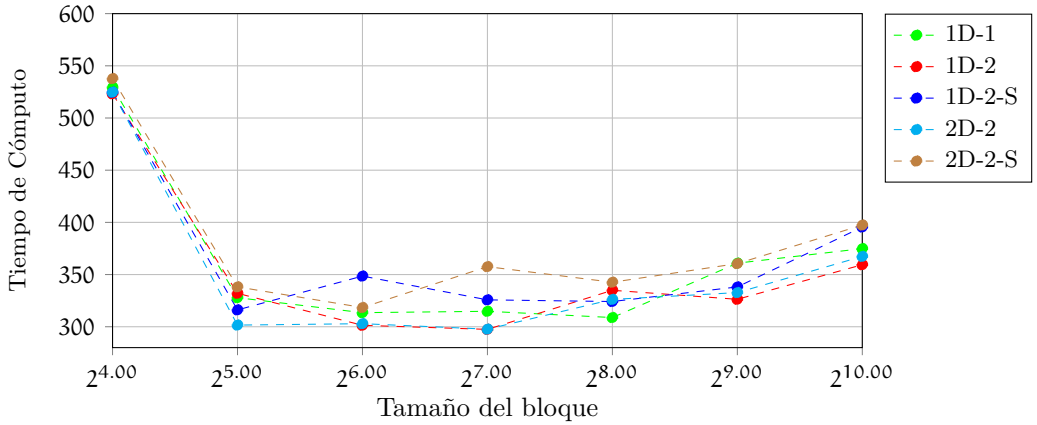


Figura 4.24: Comportamiento del tiempo de cómputo en milisegundos, obtenidos probando diferentes configuraciones de los kernels 2D en precisión doble, mostrados en la Tabla 4.6.

Para el caso 3D, la configuración ligeramente más rápida es la (1D-3) y el uso de memoria compartida, para este caso no tiene beneficio tangible (1D-3-S), para el caso de la estructura propuesta (3D-3) existe una latencia de apenas 1.07X con respecto a (1D-3) y la memoria compartida en este caso (3D-3-S) mejora el rendimiento excepto cuando se alcanza el número máximo de hilos (1024). Se puede observar una reducción del tiempo muy particular con la estructura propuesta y el uso de memoria compartida (3D-3-S), cuando el tamaño del bloque es de 512 hilos, esta se puede deber al tipo de arquitectura de la tarjeta utilizada y puede no reproducirse tal cual en otros dispositivos.

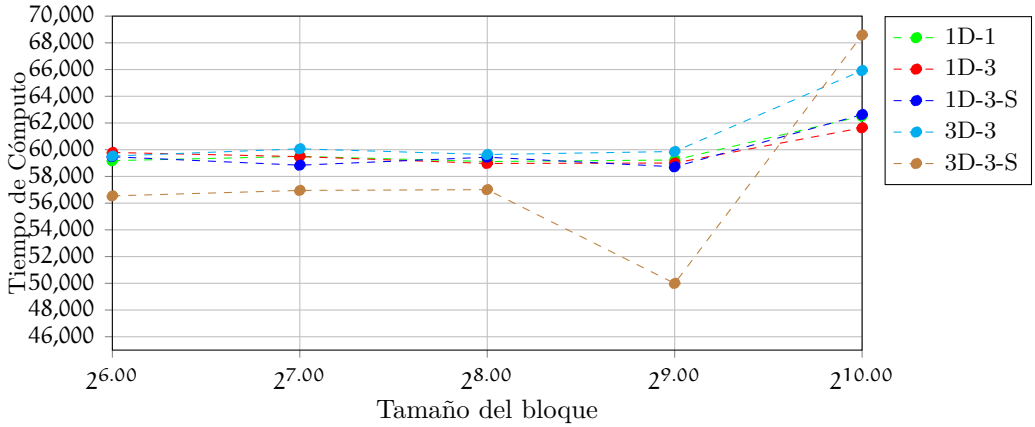


Figura 4.25: Comportamiento del tiempo de cómputo en milisegundos, obtenidos probando diferentes configuraciones de los kernels 3D en precisión sencilla mostrados en la Tabla 4.7.

Para el caso en precisión doble (`double`), se observa que se introduce una latencia de alrededor del 13 %, para los tamaños del bloque de 64, 128, 256 y nuevamente cuando llegamos al tamaño de bloque de 512 el tiempo se reduce hasta prácticamente ser igual en todos los casos, y se puede observar que el uso de la memoria compartida no tiene beneficio alguno en todos los casos.

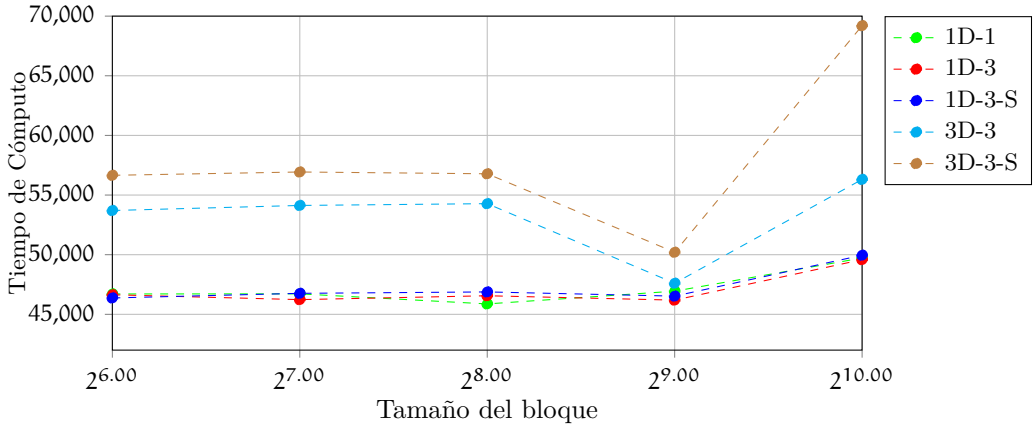


Figura 4.26: Comportamiento del tiempo de cómputo en milisegundos, obtenidos probando diferentes configuraciones de los kernels 3D en precisión doble mostrados en la Tabla 4.8

Finalmente, se comparan los tiempos de cómputo contra su contraparte secuencial con dos procesadores Intel, un Xeon E5-2630V4 y un I5-4200U. Para el caso 2D los resultados muestran que el código CUDA en la tarjeta RTX 2060 es 91,67X más rápido que el código serial corriendo en el procesador Xeon y 143,13X más rápido

con respecto al I5 (ver Figura 4.27). En el caso 3D, el código CUDA es 93,04X y 84,39X más rápido en precisión sencilla y doble respectivamente con respecto al procesador Xeon. Y con respecto al I5 128,29X y 130X veces más rápido en precisión sencilla y doble respectivamente.

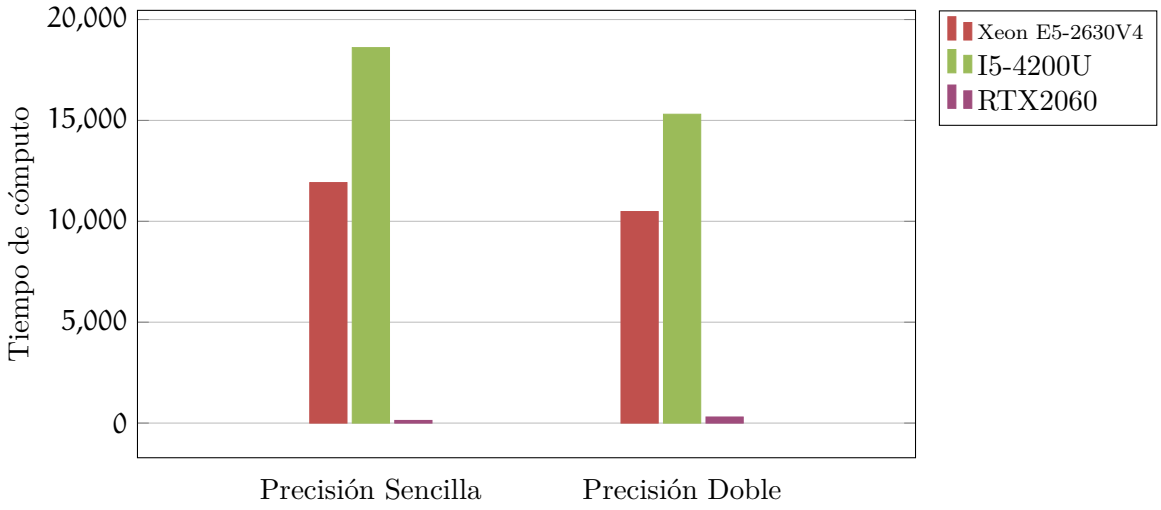


Figura 4.27: Comparación entre los tiempos de cómputo obtenidos en precisión sencilla, entre un Intel Xeon E52630V, un Intel 5I5-4200U y la tarjeta RTX2060, para el caso 2D.

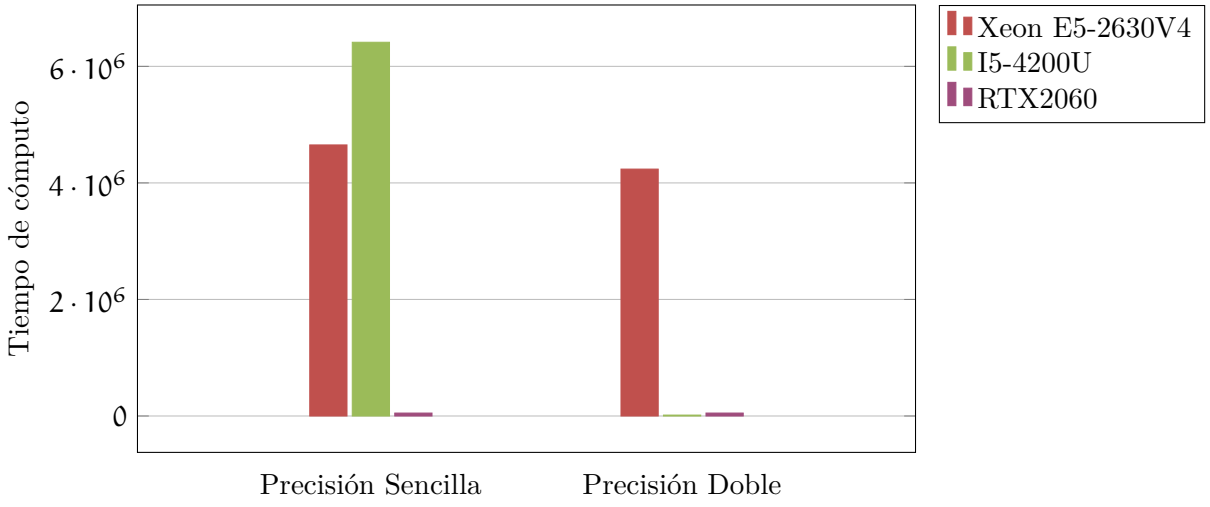


Figura 4.28: Tiempos de cómputo obtenidos en precisión sencilla, entre un Intel Xeon E52630V, un Intel 5I5-4200U y la tarjeta RTX2060, para el caso 3D.

CONCLUSIONES

Debido a que en la actualidad los GPUs se han vuelto parte indispensable de la computación ingenieril y científica, y que cada vez tienen capacidades de cómputo más poderosas, es necesario, facilitar la programación en GPU, y aunque existen herramientas basadas en directivas como OpenACC, en varias aplicaciones se prefiere migrar a bajo nivel para tener el total control del código fuente, en ese contexto, este trabajo aporta una opción en la programabilidad y legibilidad del código.

El bajo costo energético y el reducido espacio de los GPU los ha convertido en una excelente herramienta computacional para la aceleración en la ejecución de códigos. Desde su aparición en el 2007, CUDA `C` ha sido una buena opción para el cómputo científico, y se espera que esta tendencia continúe en la siguiente década, ya que la tecnología sigue en costante renovación como las tarjetas V100, que llegan a alcanzar teóricamente los 7450 GigaFlops en precisión doble y hasta 14899 GigaFlops en precisión sencilla, y son ocupadas como parte medular de Servidores y Estaciones de Trabajo dedicadas al cómputo numérico e inteligencia artificial. Esta tendencia nos conlleva a seguir buscando estrategias y estructuras de programación que faciliten la implementación de códigos en estos dispositivos.

Por otra parte el poder de los GPUs no está limitado solamente a equipos de gran escala, modelos como la Jetson AGX Xavier ofrecen en 10 cm^2 el poder de 512 núcleos de procesamiento numérico con 32 GB de memoria NVRAM, este tamaño reducido puede proporcionar a drones o satélites nuevas capacidades de procesamiento nunca antes vistas (https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit).

El aporte en este trabajo sería que la generación de código sea más legible en CUDA y menos propensos a errores, facilitando el manejo correcto de los accesos a memoria sin introducir latencia. La programación en la GPU consiste en realizar el menor número posible de transferencias entre la memoria global de la GPU y la memoria principal de la CPU. Adicionalmente la estructura desarrollada, muestra una mayor legibilidad sin sobrecargas tangibles en el tiempo de ejecución. De hecho, se espera que sirva de base para la recodificación de aplicaciones como el Cálculo del Tensor del Gradiente Gravimétrico (Couder-Castañeda et al., 2013) y el Análisis de Propagación de Ondas Electromagnéticas por medio de diferencias finitas (Rodríguez-Sánchez et al., 2018).

Con respecto a la aplicación tomada como ejemplo, la solución de la ecuación de transferencia de calor, es un excelente caso de estudio debido a que ha sido tratada

ampliamente como introducción al uso de la diferencias finitas para la solución de Ecuaciones Diferenciales Parciales. Y las diferencias finitas, proponen un desafío en la programación en la GPU, debido a la geometría de la molécula computacional, por este motivo, fue elegido este problema.

Finalmente se debe tener presente que cada avance en la arquitectura de las GPU NVIDA conlleva a una nueva versión de CUDA C, la última versión a es la 11.4.2 liberada en septiembre del 2021, 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 es necesario considerar que las primeras arquitecturas van perdiendo soporte y los nuevos compiladores dejan de generar código para las tarjetas más antiguas.

Por lo expuesto en este trabajo, se debe considerar que la programación en CUDA requiere un conocimiento técnico amplio de la arquitectura para sacar en mayor provecho posible de la tarjeta y aunque el avance de los GPUs parece promisorio en tecnología de cómputo es muy difícil predecir el futuro, por factores, como los materiales y el marketing, no obstante, es casi seguro que seguirán apareciendo plataformas basadas en la arquitectura de los GPUs. Los procesadores de nueva generación integran multiprocesadores gráficos con núcleos de procesamiento en el mismo chip, como puede verse en las nuevas arquitecturas M1 de MAC, los cuales incluyen el GPU en la misma pastilla, por lo que se necesitarán técnicas de programación que se adecuen a las nuevas tecnologías.

A

CÓDIGOS FUENTE

En este apéndice se listan los códigos fuentes desarrollados en este trabajo.

A.1 CÓDIGO FUENTE SERIAL 2D

El Listado 17, contiene el código fuente del programa serial en estándar C. Se compiló con `gcc`, de la siguiente forma `gcc main.c -O3 -lm -o 2D`, para utilizar la precisión doble debe compilarse como `gcc main.c -Ddoble -O3 -lm -o 2D`.

Listado 17: Código serial base escrito en estándar C para la difusión 2D.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#ifndef doble
typedef float real;
#else
typedef double real;
#endif

real **MATRIX_CPU(int n, int m, char const *var);
int write2D_in_time(real **M, int nx, int ny, int t);
void writematrix(real *start, int nx, int ny, int t);

int main(int argc, char *argv[]) {

  real dx, dy;
  real C = 0.5;
  real Txx, Tyy;
  real dx2;
  real dy2;
  real dt;
  int nx, ny, i, j, t = 0;
  int niter;
  int display;
```

```
  real **T, **Tn, **Temp;

  clock_t start, end;
  double cpu_time_used;

  if (argc < 5) {

    printf("Faltan parametros....\n");
    exit(1);
  }

  printf("Numero de iteraciones: %s\n", argv[1]);
  printf("Numero de elementos en x: %s\n", argv[2]);
  printf("Numero de elementos en y: %s\n", argv[3]);
  printf("Muestreo: %s\n", argv[4]);

  niter = atoi(argv[1]);
  nx = atoi(argv[2]);
  ny = atoi(argv[3]);
  display = atoi(argv[4]);

  printf("---->Numero de iteraciones: %d\n", niter);
  printf("---->Numero de elementos en x: %d\n", nx);
  printf("---->Numero de elementos en y: %d\n", ny);
  printf("---->Display: %d", display);
  dx = 1.0 / ((real)nx - 1.0);
  dy = 1.0 / ((real)ny - 1.0);

  dx2 = dx * dx;
  dy2 = dy * dy;
  dt = (dx2 * dy2) / (2 * C * (dx2 + dy2));

  printf("\ndx:%f dy:%f dt:%f", dx, dy, dt);

  // getchar();

  T = MATRIX_CPU(ny, nx, "T");
  Tn = MATRIX_CPU(ny, nx, "Tn");

  for (i = 0; i < ny; i++)
    for (j = 0; j < nx; j++) {

#ifndef doble

      if ((powf(((real)j + 1.0f) * dx - 0.5f, 2.0f) +
           powf(((real)i + 1.0f) * dy - 0.5f, 2.0f) <=
          0.1f) &&
```

```
          (powf(((real)j + 1.0f) * dx - 0.5f, 2.0f) +
               powf(((real)i + 1.0f) * dy - 0.5f, 2.0f) >=
           .05f))
        T[i][j] = 1.0f;
      else
        T[i][j] = 0.0f;

#else

      if ((pow(((real)j + 1.0) * dx - 0.5, 2.0) +
               pow(((real)i + 1.0) * dy - 0.5, 2.0) <=
           0.1) &&
          (pow(((real)j + 1.0) * dx - 0.5, 2.0) +
               pow(((real)i + 1.0) * dy - 0.5, 2.0) >=
           .05))
        T[i][j] = 1.0;
      else
        T[i][j] = 0.0;

#endif
    }

  printf("\nTermine");
  // getchar();
  printf("\nVamos a escribir la condiciones iniciales...");
  fflush(stdout);
  write2D_in_time(T, ny, nx, t);
  // writematrix(&(T[0][0]), ny, nx, t);
  printf("Presione cualquier tecla...");
  // getchar();

  printf("\n%d", t);

  start = clock();
  while (t < niter) {
    for (i = 1; i < (ny - 1); i++)
      for (j = 1; j < (nx - 1); j++) {
        Txx = (T[i][j + 1] - 2.0 * T[i][j] + T[i][j - 1]) / dx2
            ;
        Tyy = (T[i + 1][j] - 2.0 * T[i][j] + T[i - 1][j]) / dy2
            ;
        Tn[i][j] = T[i][j] + dt * C * (Txx + Tyy);
      }
    t++;

    if (t % display == 0) {
      printf("\nt:%d", t);
```

```
120       write2D_in_time(Tn, ny, nx, t);
          // writematrix(&(T[0][0]),ny,nx,t);
        }
        Temp = T;
        T = Tn;
125     Tn = Temp;
      }
      end = clock();

      cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
130
      printf("\nTiempo Utilizado: %lf", cpu_time_used);

      return 0;
    }
135
    real **MATRIX_CPU(int n, int m, char const *var) {
      real **A = NULL;
      int i, j;

140   printf("\nAllocating matrix:  %s of total size %d", var, n *
          m);
      // calloc((void***)&A, n*sizeof(real*));
      A = (real **)calloc(n, sizeof(real *));
      if (A == NULL)
        puts("Memory problem calloc first level"), exit(-1);
145   // calloc((void**)&A[0], n*m*sizeof(real));
      A[0] = (real *)calloc(n * m, sizeof(real));
      if (A[0] == NULL)
        puts("Memory second level"), exit(-1);
      for (i = 1; i < n; i++)
150     A[i] = A[0] + i * m;
      for (i = 0; i < n; i++)
        for (j = 0; j < m; j++)
          A[i][j] = 0.0;
      return A;
155 }

    int write2D_in_time(real **M, int ny, int nx, int t) {
    #ifndef doble
      char dir[60] = "/home/ccouder/tmp/s-2DS_";
160 #else
      char dir[60] = "/home/ccouder/tmp/2DS_";
    #endif
      char num[6];
      char ext[] = ".dat";
165   int i, j;
```

```
  FILE *f;
  sprintf(num, "%d", t);
  strcat(dir, num);
  strcat(dir, ext);
  printf("File :%s\n", dir);
  f = fopen(dir, "w");
  for (i = 0; i < ny; i++) {
    for (j = 0; j < nx; j++) {
#ifndef doble
      fprintf(f, "%.8f\t", M[i][j]);
#else
      fprintf(f, "%.15lf\t", M[i][j]);
#endif
    }
    fprintf(f, "\n");
  }
  fflush(f);
  fclose(f);
  return 0;
}

void writematrix(real *start, int nx, int ny, int t) {
#ifndef doble
  char dir[60] = "/home/ccouder/tmp/s-2DS_";
#else
  char dir[60] = "/home/ccouder/tmp/2DS_";
#endif
  char num[6];
  char ext[] = ".bin";
  int i, j;

  sprintf(num, "%d", t);
  strcat(dir, num);
  strcat(dir, ext);

  printf("File :%s\n", dir);

  FILE *file = fopen(dir, "wb");

  fwrite(start, nx * ny * sizeof(real), 1, file);

  fclose(file);
}
```

Para ejecutar el anterior código deberán pasarse los parámetros, número de iteraciones, tamaño de la malla en puntos en `x` y `y` y el número de iteraciones en las que se llevará a cabo la escritura de archivos, por ejemplo si escribimos `./2D 20000`

`301 201 500`, se ejecutarán 20,000 iteraciones en tiempo, se trabajará con una malla de `301 x 201` y se escribirán a disco los resultados cada 500 iteraciones. El programa programa produce los archivos de texto numerados como `0.dat`, `500.dat`, `1000.dat`, `1500.dat`, `2000.dat` y así sucesivamente hasta llegar a `20000.dat`, los cuales contienen una matriz de texto que contienen los resultados de las iteraciones en el tiempo.

Los archivos son fácilmente cargables desde cualquier programa de graficación como `MATLAB` mediante los comandosload 0.dat, `contourf(X0,25)`. El archivo `0.dat` contiene las condiciones iniciales.

A.2 CÓDIGO FUENTE SERIAL 3D

El Listado 18 contiene el código fuente del programa serial en estándar C. Se compiló con `gcc ver 10` con el siguiente comando `gcc-10 main.c -O3 -lm -3D`.

Listado 18: Código serial base escrito en estándar C para la difusión 2D.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#ifndef doble
typedef float real;
#else
typedef double real;
#endif

int write3D(real ***M, int ny, int nx, int nz);
void writeX(int nx, int ny, real dx);
real ***CUBE_CPU(int n, int m, int z, char const *var);
int write3D_in_time(real ***M, int nx, int ny, int nz, int t);

int main(int argc, char *argv[]) {
  real dx, dy, dz;
  real C = 0.5;
  real dx2 = dx * dx;
  real dy2 = dy * dy;
  real dz2 = dz * dz;
  real Txx, Tyy, Tzz;
  int t = 0;
  real dt;
  int nx, ny, nz, i, j, k;
```

```
  real ***T, ***Tn, ***Temp;

  int niter;
  int display;

  clock_t start, end;
  double cpu_time_used;

  if (argc < 6) {
    printf("Falta numero de iteraciones\n");
    exit(1);
  }

  printf("Numero de iteraciones: %s\n", argv[1]);
  printf("Numero de elementos en x: %s\n", argv[2]);
  printf("Numero de elementos en y: %s\n", argv[3]);
  printf("Numero de elementos en z: %s\n", argv[4]);
  printf("Muestreo: %s\n", argv[5]);

  niter = atoi(argv[1]);
  nx = atoi(argv[2]);
  ny = atoi(argv[3]);
  nz = atoi(argv[4]);
  display = atoi(argv[5]);

  printf("Numero de iteraciones: %d\n", niter);
  printf("Numero de elementos en x: %d\n", nx);
  printf("Numero de elementos en y: %d\n", ny);
  printf("Numero de elementos en z: %d\n", nz);
  printf("Muestreo: %d\n", display);

  dx = 1.0 / ((real)nx - 1.0);
  dy = 1.0 / ((real)ny - 1.0);
  dz = 1.0 / ((real)nz - 1.0);

  dx2 = dx * dx;
  dy2 = dy * dy;
  dz2 = dz * dz;

  dt = (dx2 * dy2 * dz2) / (2.0 * C * (dx2 * dy2 + dy2 * dz2 +
     dx2 * dz2));

  printf("\nnx:%f ny:%f nz:%f dt:%f", dx, dy, dz, dt);
  getchar();

  T = CUBE_CPU(ny, nx, nz, "T");
  Tn = CUBE_CPU(ny, nx, nz, "Tn");
```

```
  for (i = 0; i < ny; i++)
    for (j = 0; j < nx; j++)
      for (k = 0; k < nz; k++) {
#ifndef doble
        if ((powf(((float)j + 1.0) * dx - 0.5, 2.0) +
                  powf(((float)i + 1.0) * dy - 0.5, 2.0) +
                  powf(((float)k + 1.0) * dz - 0.5, 2.0) <=
             0.1) &&
            (powf(((float)j + 1.0) * dx - 0.5, 2.0) +
                  powf(((float)i + 1.0) * dy - 0.5, 2.0) +
                  powf(((float)k + 1.0) * dz - 0.5, 2.0) >=
             .05))
          T[i][j][k] = 1.0;
        else
          T[i][j][k] = 0.0;

#else

        if ((pow(((double)j + 1.0) * dx - 0.5, 2.0) +
                  pow(((double)i + 1.0) * dy - 0.5, 2.0) +
                  pow(((double)k + 1.0) * dz - 0.5, 2.0) <=
             0.1) &&
            (pow(((double)j + 1.0) * dx - 0.5, 2.0) +
                  pow(((double)i + 1.0) * dy - 0.5, 2.0) +
                  pow(((double)k + 1.0) * dz - 0.5, 2.0) >=
             .05))
          T[i][j][k] = 1.0;
        else
          T[i][j][k] = 0.0;

#endif
      }

  write3D_in_time(T, ny, nx, nz, t);
  printf("\nEscribi las condiciones iniciales...");
  start = clock();
  while (t < niter) {
    for (i = 1; i < (ny - 1); i++)
      for (j = 1; j < (nx - 1); j++)
        for (k = 1; k < (nz - 1); k++) {
          Tzz = (T[i][j][k + 1] - 2.0 * T[i][j][k] + T[i][j][k
             - 1]) / dz2;
          Txx = (T[i][j + 1][k] - 2.0 * T[i][j][k] + T[i][j -
             1][k]) / dx2;
          Tyy = (T[i + 1][j][k] - 2.0 * T[i][j][k] + T[i - 1][j
             ][k]) / dy2;
```

```
          Tn[i][j][k] = T[i][j][k] + dt * C * (Txx + Tyy + Tzz)
              ;
        }
    t++;

    if (t % display == 0) {
      printf("\n%d", t);
      write3D_in_time(Tn, ny, nx, nz, t);
    }

    Temp = T;
    T = Tn;
    Tn = Temp;
  }
  end = clock();
  cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
  printf("\nTiempo Utilizado: %f", cpu_time_used);
  return 0;
}

real ***CUBE_CPU(int n, int m, int z, char const *var) {
  real ***A = NULL;

  int i, j, k;

  printf("\nAllocating a cube:  %s of total size %d", var, n *
     m * z);

  A = (real ***)calloc(n, sizeof(real **));

  if (A == NULL)
    puts("Memory problem calloc first level"), exit(-1);
  A[0] = (real **)calloc(n * m, sizeof(real *));

  for (i = 1; i < n; i++)
    A[i] = A[0] + i * m;

  A[0][0] = (real *)calloc(n * m * z, sizeof(real));

  for (j = 1; j < (n * m); j++)
    A[0][j] = A[0][0] + j * z;

  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      for (k = 0; k < z; k++) {
        A[i][j][k] = 0.0;
```

```
      }

  return A;
}

int write3D_in_time(real ***M, int ny, int nx, int nz, int t) {
#ifndef doble
  char dir[60] = "/home/ccouder/tmp/s-3DS";
#else
  char dir[60] = "/home/ccouder/tmp/3DS";
#endif

  char num[6];
  char ext[] = ".dat";
  int i, j, k;
  FILE *f;

  sprintf(num, "%d", t);
  strcat(dir, num);
  strcat(dir, ext);

  printf("File :%s", dir);
  f = fopen(dir, "w");

  for (i = 0; i < ny; i++)
    for (j = 0; j < nx; j++)
      for (k = 0; k < nz; k++) {
#ifndef doble
        fprintf(f, "%d %d %d %.8f\n", i + 1, j + 1, k + 1, M[i
           ][j][k]);
#else
        fprintf(f, "%d %d %d %.15lf\n", i + 1, j + 1, k + 1, M[
           i][j][k]);
#endif
      }
  fflush(f);
  fclose(f);
  return 0;
}
```

A.3 KERNELS 2D

A.3.1 *Kernel 2D-1-1*

En el Listado 19 se muestra el código completo en CUDA C para el caso 2D, utilizando un arreglo unidimensional y un solo índice l. Se compila como `nvcc 2D-1-1.cu`, si se requiere el precisión doble se agrega la bandear `-Ddoble` y para modificar el tamaño del `warp` se usa la bandera `Dtilex`, por ejemplo, si se requiere precisión doble con un `warp` de tamaño 64, se compila como: `nvcc 2D-1-1.cu -Ddoble -Dtilex=64`.

Listado 19: Código CUDA para el caso 2D, con un arreglo unidimensional y un índice.

```
#include <cuda.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef doble
typedef float real;
#else
typedef double real;
#endif

real **MATRIX_CPU(int n, int m, char const *var);
int write2D_in_time(real *M, int nx, int ny, int t);
void writematrix(real *start, int nx, int ny, int t);

#ifndef tilex
#define tilex 16
#endif

__global__ void kernel(real *T, real *Tn, int nx, int ny, real
   dt, real C,
                       real dx2, real dy2) {
  int l;
  l = blockIdx.x * blockDim.x + threadIdx.x;
  real Txx;
  real Tyy;
  // if(l>ny && l<((nx*ny-1)-ny) && !((l%ny)==0 || ((l+1)%ny ==
      0)))

  if (l > nx && l < ((nx * ny - 1) - nx) &&
      !((l % nx) == 0 || ((l + 1) % nx == 0))) {
    Txx = (T[l + 1] - 2.0 * T[l] + T[l - 1]) / dx2;
    Tyy = (T[l + nx] - 2.0 * T[l] + T[l - nx]) / dy2;
```

```
    Tn[l] = T[l] + dt * C * (Txx + Tyy);
  }
}

int main(int argc, char *argv[]) {
  real dx, dy;

#ifndef doble
  float C = 0.5f;
#else
  double C = 0.5;
#endif

  real dx2;
  real dy2;
  real dt;
  int nx, ny, i, j, t = 0;
  real *Tdev, *Thost, *Tdev_n, *Temp;
  int display;
  int niter;
  //----------------------------
  int numberofdevices, dev;
  cudaDeviceProp prop;
  int blockSize;
  int gridSize;

  //----------------------------

  cudaEvent_t start, stop;
  float elapsedTime;

  // nx = (int)(1.0f/dx)+1;
  // ny = (int)(1.0f/dy)+1;

  if (argc < 5) {

    printf("Falta el numero de iteraciones\n");
    exit(1);
  }

  printf("Numero de iteraciones: %s\n", argv[1]);
  printf("Numero de elementos en x: %s\n", argv[2]);
  printf("Numero de elementos en y: %s\n", argv[3]);
  printf("Muestreo: %s\n", argv[4]);

  niter = atoi(argv[1]);
  nx = atoi(argv[2]);
```

```
80  ny = atoi(argv[3]);
    display = atoi(argv[4]);

    printf("-->Numero de iteraciones: %d\n", niter);
    printf("-->Numero de elementos en x: %d\n", nx);
85  printf("-->Numero de elementos en y: %d\n", ny);
    printf("-->Display: %d", display);

    dx = 1.0 / ((real)nx - 1.0);
    dy = 1.0 / ((real)ny - 1.0);
90
    dx2 = dx * dx;
    dy2 = dy * dy;
    dt = (dx2 * dy2) / (2 * C * (dx2 + dy2));

95  printf("\ndx:%f dy:%f dt:%f", dx, dy, dt);

    getchar();
    cudaGetDeviceCount(&numberofdevices);

100 printf("\nNumber of devices: %d", numberofdevices);

    for (i = 0; i < numberofdevices; i++) {
      cudaGetDeviceProperties(&prop, i);
      printf("\n------- %d ----------", i);
105   printf("\n\t Name: %s", prop.name);
      printf("\n\t Compute: %d.%d", prop.major, prop.minor);
      printf("\n\t Multiprocessor: %d", prop.multiProcessorCount)
          ;
      printf("\n\t Total global mem: %ld \n", (long int)prop.
          totalGlobalMem);
    }
110 cudaSetDevice(2);
    cudaGetDevice(&dev);
    printf("\nCurrent device:%d", dev);
    cudaMallocHost((void **)&Thost, nx * ny * sizeof(real));
    cudaMalloc((void **)&Tdev, nx * ny * sizeof(real));
115 cudaMalloc((void **)&Tdev_n, nx * ny * sizeof(real));

    cudaMemcpy(Tdev, Thost, nx * ny * sizeof(real),
       cudaMemcpyHostToDevice);
    cudaMemcpy(Tdev_n, Thost, nx * ny * sizeof(real),
       cudaMemcpyHostToDevice);

120 for (i = 0; i < ny; i++)
      for (j = 0; j < nx; j++) {
```

```
#ifndef doble

      if ((powf(((float)j + 1.0f) * dx - 0.5f, 2.0f) +
                powf(((float)i + 1.0f) * dy - 0.5f, 2.0f) <=
           0.1f) &&
          (powf(((float)j + 1.0f) * dx - 0.5f, 2.0f) +
                powf(((float)i + 1.0f) * dy - 0.5f, 2.0f) >=
           .05f))
        Thost[i * nx + j] = 1.0f;
      else
        Thost[i * nx + j] = 0.0f;

#else

      if ((pow(((double)j + 1.0) * dx - 0.5, 2.0) +
                pow(((double)i + 1.0) * dy - 0.5, 2.0) <=
           0.1) &&
          (pow(((double)j + 1.0) * dx - 0.5, 2.0) +
                pow(((double)i + 1.0) * dy - 0.5, 2.0f) >=
           .05)) {
        printf("%d  %d\n", i, j);
        Thost[i * nx + j] = 1.0;
      } else
        Thost[i * nx + j] = 0.0;

#endif
    }

  cudaMemcpy(Tdev, Thost, nx * ny * sizeof(real),
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU
  cudaMemcpy(Tdev_n, Thost, nx * ny * sizeof(real),
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU

  cudaMemcpy(Thost, Tdev, nx * ny * sizeof(real),
             cudaMemcpyDeviceToHost); // Transferir a la
                memoria del GPU

  write2D_in_time(Thost, ny, nx, t);
  writematrix(&(Thost[0]), ny, nx, t);

  getchar();
  blockSize = tilex;
  gridSize = (int)ceil((real)(nx * ny) / blockSize);

  printf("blockSize: %d, gridSize: %d", blockSize, gridSize);
```

```
  getchar();

  // Medir el tiempo
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);
  while (t < niter) {
    kernel<<<gridSize, blockSize>>>(Tdev, Tdev_n, nx, ny, dt, C
        , dx2, dy2);
    t++;

    if (t % display == 0) {
      printf("\nt:%d", t);
      cudaMemcpy(Thost, Tdev_n, nx * ny * sizeof(real),
          cudaMemcpyDeviceToHost);
      cudaDeviceSynchronize();
      write2D_in_time(Thost, ny, nx, t);
      writematrix(&(Thost[0]), ny, nx, t);
    }

    Temp = Tdev;
    Tdev = Tdev_n;
    Tdev_n = Temp;
  }
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&elapsedTime, start, stop);

  printf("\nElapsed time: %f\n", elapsedTime);
  return 0;
}

int write2D_in_time(real *M, int ny, int nx, int t) {
#ifndef doble
  char dir[60] = "/home/ccouder/tmp/s-2D-1-1_";
#else
  char dir[60] = "/home/ccouder/tmp/2D-1-1_";
#endif

  char num[6];
  char ext[] = ".dat";
  int i, j;
  FILE *f;

  sprintf(num, "%d", t);
  strcat(dir, num);
  strcat(dir, ext);
```

```
  printf("\nArchivo:%s\n", dir);

  f = fopen(dir, "w");
  for (i = 0; i < ny; i++) {
    for (j = 0; j < nx; j++) {
#ifndef doble
      fprintf(f, "%.8f\t", M[i * nx + j]);
#else
      fprintf(f, "%.15lf\t", M[i * nx + j]);
#endif
    }
    fprintf(f, "\n");
  }
  fflush(f);
  fclose(f);
  return 0;
}

void writematrix(real *start, int nx, int ny, int t) {
#ifndef doble
  char dir[60] = "/home/ccouder/tmp/s-2D-1-1_";
#else
  char dir[60] = "/home/ccouder/tmp/2D-1-1_";
#endif

  char num[6];
  char ext[] = ".bin";

  // printf("\nDirectorio:%s\n", dir);
  sprintf(num, "%d", t);
  strcat(dir, num);
  strcat(dir, ext);

  printf("File :%s\n", dir);

  FILE *file = fopen(dir, "wb");

  fwrite(start, nx * ny * sizeof(real), 1, file);

  fclose(file);
}
```

A.3.2 *Kernel 2D-1-2*

En el Listado 20 se muestra el código que contiene los `kernels` para el caso de un arreglo unidimensional y dos índices, con y sin el uso de memoria compartida. Las banderas de compilación son: `shared`, `double`, `tilex` y `tiley`. Por ejemplo, si deseamos utilizar memoria compartida, y crear un `warp` de `8 x 8`, y además utilizar precisión doble, deberemos compilar como: `nvcc -Dshared -Ddoble -Dtilex=8 -Dtiley=8`.

Listado 20: Código CUDA para el caso 2D, con un arreglo unidimensional y dos índices.

```
#include <cuda.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef doble
typedef float real;
#else
typedef double real;
#endif

real **MATRIX_CPU(int n, int m, char const *var);
int write2D_in_time(real *M, int ny, int nx, int t);
void writematrix(real *start, int nx, int ny, int t);

#ifndef tilex
#define tilex 8
#endif

#ifndef tiley
#define tiley 8
#endif

__global__ void kernel(real *T, real *Tn, int ny, int nx, real
    dt, real C,
                       real dx2, real dy2) {
  // nx number of elements in the y direction
  // ny number of elements in the x direction
  int i, j, l;
  j = blockIdx.x * blockDim.x + threadIdx.x;
  i = blockIdx.y * blockDim.y + threadIdx.y;

  real Txx;
  real Tyy;
```

```
35
  if ((j > 0) && (j < (nx - 1)) && (i > 0) && (i < (ny - 1))) {
    l = i * nx + j;
    Txx = (T[l + 1] - 2.0 * T[l] + T[l - 1]) / dx2;
    Tyy = (T[l + nx] - 2.0 * T[l] + T[l - nx]) / dy2;
40  Tn[l] = T[l] + dt * C * (Txx + Tyy);
  }
}

__global__ void kernel_shared(real *T, real *Tn, int ny, int nx
    , real dt,
45                              real C, real dx2, real dy2) {

  int i, j, l, m, index;
  real Txx;
  real Tyy;
50
  __shared__ real tile[tiley + 2][tilex + 2];
  __shared__ real tilen[tiley][tilex];

  j = blockIdx.x * blockDim.x + threadIdx.x;
55 i = blockIdx.y * blockDim.y + threadIdx.y;
  index = i * nx + j;
  if ((i >= 0) && (i <= (ny - 1)) && (j >= 0) && (j <= (nx - 1)
     )) {
    tile[threadIdx.y + 1][threadIdx.x + 1] = T[index];
  }
60
  if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1))) {
    if (threadIdx.x == (blockDim.x - 1))
      tile[threadIdx.y + 1][threadIdx.x + 2] = T[index + 1]; //
          right
    if (threadIdx.y == (blockDim.y - 1))
65    tile[threadIdx.y + 2][threadIdx.x + 1] = T[index + nx];
        // down
    if (threadIdx.x == (0))
      tile[threadIdx.y + 1][threadIdx.x] = T[index - 1]; //
        left
    if (threadIdx.y == (0))
      tile[threadIdx.y][threadIdx.x + 1] = T[index - nx]; // up
70 }
  __syncthreads();

  if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1))) {
    m = threadIdx.x + 1;
75  l = threadIdx.y + 1;
```

```
    Txx = (tile[l][m + 1] - 2.0 * tile[l][m] + tile[l][m - 1])
        / dx2;
    Tyy = (tile[l + 1][m] - 2.0 * tile[l][m] + tile[l - 1][m])
        / dy2;
    tilen[l - 1][m - 1] = tile[l][m] + dt * C * (Txx + Tyy);
  }
  if (j == 0)
    // tilen[threadIdx.y][threadIdx.x] = tile[threadIdx.y+1][
        threadIdx.x+1];
    tilen[threadIdx.y][threadIdx.x] = 0.0; // Condicion
        Dirchetl
  if (j == (nx - 1))
    // tilen[threadIdx.y][threadIdx.x] = tile[threadIdx.y+1][
        threadIdx.x+1];
    tilen[threadIdx.y][threadIdx.x] = 0.0;
  if (i == 0)
    // tilen[threadIdx.y][threadIdx.x] = tile[threadIdx.y+1][
        threadIdx.x+1];
    tilen[threadIdx.y][threadIdx.x] = 0.0;
  if (i == (ny - 1))
    // tilen[threadIdx.y][threadIdx.x] = tile[threadIdx.y+1][
        threadIdx.x+1];
    tilen[threadIdx.y][threadIdx.x] = 0.0;

  if ((i >= 0) && (i <= (ny - 1)) && (j >= 0) && (j <= (nx - 1)
      ))
    Tn[index] = tilen[threadIdx.y][threadIdx.x];
}

int main(int argc, char *argv[]) {
  real dx, dy;

#ifndef doble
  float C = 0.5f;
#else
  double C = 0.5;
#endif

  real dx2;
  real dy2;
  real dt;
  int nx, ny, i, j, t = 0;
  real *Tdev, *Thost, *Tdev_n, *Temp;
  //-----------------------------
  int numberofdevices, dev;
  cudaDeviceProp prop;
  // int blockSize;
```

```
115   // int gridSize;
      dim3 grid, block;
      //----------------------------
      cudaEvent_t start, stop;
      float elapsedTime;
120   int niter;
      int display;

      if (argc < 5) {

125     printf("Faltan parametros\n");
        exit(1);
      }

      printf("Numero de iteraciones: %s\n", argv[1]);
130   printf("Numero de elementos en x: %s\n", argv[2]);
      printf("Numero de elementos en y: %s\n", argv[3]);
      printf("Muestreo: %s\n", argv[4]);

      niter = atoi(argv[1]);
135   nx = atoi(argv[2]);
      ny = atoi(argv[3]);
      display = atoi(argv[4]);

      printf("---->Numero de iteraciones: %d\n", niter);
140   printf("---->Numero de elementos en x: %d\n", nx);
      printf("---->Numero de elementos en y: %d\n", ny);
      printf("---->Display: %d\n", display);

      dx = 1.0 / ((real)nx - 1.0);
145   dy = 1.0 / ((real)ny - 1.0);

      dx2 = dx * dx;
      dy2 = dy * dy;
      dt = (dx2 * dy2) / (2 * C * (dx2 + dy2));
150
      printf("\ndx:%f dy:%f dt:%f", dx, dy, dt);

      getchar();

155   block.x = tilex;
      block.y = tiley;

      // si el bloque es de 8x8 luego entonces 64 threads!

160   // nx = (int)(1.0f/dx)+1;
      // ny = (int)(1.0f/dy)+1;
```

```
  cudaGetDeviceCount(&numberofdevices);
  cudaSetDevice(2);
  cudaGetDevice(&dev);
  printf("\nNumber of devices: %d", numberofdevices);

  for (i = 0; i < numberofdevices; i++) {
    cudaGetDeviceProperties(&prop, i);
    printf("\n------- %d ----------", i);
    printf("\n\t Name: %s", prop.name);
    printf("\n\t Compute: %d.%d", prop.major, prop.minor);
    printf("\n\t Multiprocessor: %d", prop.multiProcessorCount)
        ;
    printf("\n\t Total global mem: %d \n", (long int)prop.
       totalGlobalMem);
  }
  printf("\nCurrent device:%d", dev);

  cudaMallocHost((void **)&Thost, nx * ny * sizeof(real));
  cudaMalloc((void **)&Tdev, nx * ny * sizeof(real));
  cudaMalloc((void **)&Tdev_n, nx * ny * sizeof(real));

  cudaMemcpy(Tdev, Thost, nx * ny * sizeof(real),
     cudaMemcpyHostToDevice);
  cudaMemcpy(Tdev_n, Thost, nx * ny * sizeof(real),
     cudaMemcpyHostToDevice);

  for (i = 0; i < ny; i++)
    for (j = 0; j < nx; j++) {

#ifndef doble

      if ((powf(((float)j + 1.0f) * dx - 0.5f, 2.0f) +
           powf(((float)i + 1.0f) * dy - 0.5f, 2.0f) <=
          0.1f) &&
          (powf(((float)j + 1.0f) * dx - 0.5f, 2.0f) +
           powf(((float)i + 1.0f) * dy - 0.5f, 2.0f) >=
          .05f))
        Thost[i * nx + j] = 1.0f;
      else
        Thost[i * nx + j] = 0.0f;

#else

      if ((pow(((double)j + 1.0) * dx - 0.5, 2.0) +
           pow(((double)i + 1.0) * dy - 0.5, 2.0) <=
          0.1) &&
```

```
205           (pow(((double)j + 1.0) * dx - 0.5, 2.0) +
                   pow(((double)i + 1.0) * dy - 0.5, 2.0f) >=
               .05)) {
          printf("%d  %d\n", i, j);
          Thost[i * nx + j] = 1.0;
210       } else
          Thost[i * nx + j] = 0.0;

#endif
    }
215
  cudaMemcpy(Tdev, Thost, nx * ny * sizeof(real),
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU
  cudaMemcpy(Tdev_n, Thost, nx * ny * sizeof(real),
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU
220
  cudaMemcpy(Thost, Tdev, nx * ny * sizeof(real),
             cudaMemcpyDeviceToHost); // Transferir a la
                memoria del GPU
  write2D_in_time(Thost, ny, nx, t);  // regreso de la memoria
     ....
  writematrix(&(Thost[0]), ny, nx, t);
225
  getchar();
  // blockSize = 32;
  grid.x = (int)ceil((real)(nx) / block.x);
  grid.y = (int)ceil((real)(ny) / block.y);
230
  printf("block.x: %d block.y: %d\n", block.x, block.y);
  printf("grid.x: %d grid.y: %d\n", grid.x, grid.y);
  getchar();
  // Medir el tiempo
235  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);
  //-----------------
  while (t < niter) {
240// Llamar al kernel
#ifdef SHARED
    kernel_shared<<<grid, block>>>(Tdev, Tdev_n, ny, nx, dt, C,
        dx2, dy2);
#else
    kernel<<<grid, block>>>(Tdev, Tdev_n, ny, nx, dt, C, dx2,
       dy2);
245#endif
```

```
    t++;

    if (t % display == 0) {
      printf("\nt:%d", t);
      cudaMemcpy(Thost, Tdev_n, nx * ny * sizeof(real),
          cudaMemcpyDeviceToHost);
      cudaDeviceSynchronize();
      write2D_in_time(Thost, ny, nx, t);
      writematrix(&(Thost[0]), ny, nx, t);
    }

    Temp = Tdev;
    Tdev = Tdev_n;
    Tdev_n = Temp;
  }
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&elapsedTime, start, stop);

  printf("\nElapsed time: %f\n", elapsedTime);
  return 0;
}

int write2D_in_time(real *M, int ny, int nx, int t) {
#ifndef doble
  char dir[60] = "/home/ccouder/tmp/s-2D-1-2_";
#else
  char dir[60] = "/home/ccouder/tmp/2D-1-2_";
#endif

  char num[6];
#ifndef SHARED
  char ext[] = ".dat";
#else
  char ext[] = "S.dat";
#endif
  int i, j;
  FILE *f;

  sprintf(num, "%d", t);
  strcat(dir, num);
  strcat(dir, ext);

  printf("\nArchivo:%s\n", dir);

  f = fopen(dir, "w");
```

```
  for (i = 0; i < ny; i++) {
    for (j = 0; j < nx; j++) {
#ifndef doble
      fprintf(f, "%.8f\t", M[i * nx + j]);
#else
      fprintf(f, "%.15lf\t", M[i * nx + j]);
#endif
    }
    fprintf(f, "\n");
  }
  fflush(f);
  fclose(f);
  return 0;
}

void writematrix(real *start, int nx, int ny, int t) {
#ifndef doble
  char dir[60] = "/home/ccouder/tmp/s-2D-1-2_";
#else
  char dir[60] = "/home/ccouder/tmp/2D-2-2_";
#endif
  char num[6];
#ifndef SHARED
  char ext[] = ".bin";
#else
  char ext[] = "S.bin";
#endif

  // printf("\nDirectorio:%s\n", dir);
  sprintf(num, "%d", t);
  strcat(dir, num);
  strcat(dir, ext);

  printf("File :%s\n", dir);

  FILE *file = fopen(dir, "wb");

  fwrite(start, nx * ny * sizeof(real), 1, file);

  fclose(file);
}
```

A.3.3 *Kernel 2D-2-2*

En el Listado 21 se muestra el código que contiene los `kernels` para el caso de un arreglo bidimensional y dos índices. Las banderas de compilación son: `shared` , `double`, `tilex` y `tiley`. Por ejemplo, si deseamos utilizar memoria compartida, y crear un `warp` de `8 x 8`, y además utilizar precisión doble, deberemos compilar como: `nvcc -Dshared -Ddoble -Dtilex=8 -Dtiley=8`.

Listado 21: Código CUDA para el caso 2D, con un arreglo bidimensional y dos índices.

```
#include <cuda.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef doble
typedef float real;
#else
typedef double real;
#endif

void MATRIX_GPU(real **P, real ***M, int n, int m, char const *
    var);
real **MATRIX_CPU(int n, int m, char const *var);
int write2D_in_time(real *M, int ny, int nx, int t);
void writematrix(real *start, int nx, int ny, int t);

#ifndef tilex
#define tilex 8
#endif

#ifndef tiley
#define tiley 8
#endif

__global__ void kernel(real **T, real **Tn, int ny, int nx,
    real dt, real C,
                       real dx2, real dy2) {
  int i, j;
  j = blockIdx.x * blockDim.x + threadIdx.x;
  i = blockIdx.y * blockDim.y + threadIdx.y;

  real Txx;
  real Tyy;
```

```
35    if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1))) {
        Txx = (T[i][j + 1] - 2.0 * T[i][j] + T[i][j - 1]) / dx2;
        Tyy = (T[i + 1][j] - 2.0 * T[i][j] + T[i - 1][j]) / dy2;
        Tn[i][j] = T[i][j] + dt * C * (Txx + Tyy);
      }
40  }

  __global__ void kernel_shared(real **T, real **Tn, int ny, int
      nx, real dt,
                                real C, real dx2, real dy2) {
      int i, j, l, m;
45    real Txx;
      real Tyy;

      __shared__ real tile[tiley + 2][tilex + 2];
      __shared__ real tilen[tiley][tilex];
50
      j = blockIdx.x * blockDim.x + threadIdx.x;
      i = blockIdx.y * blockDim.y + threadIdx.y;

      if ((i >= 0) && (i <= (ny - 1)) && (j >= 0) && (j <= (nx - 1)
          ))
55      tile[threadIdx.y + 1][threadIdx.x + 1] = T[i][j];

      if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1))) {
        if (threadIdx.x == (blockDim.x - 1))
          tile[threadIdx.y + 1][threadIdx.x + 2] = T[i][j + 1]; //
              right
60      if (threadIdx.y == (blockDim.y - 1))
          tile[threadIdx.y + 2][threadIdx.x + 1] = T[i + 1][j]; //
              down
        if (threadIdx.x == (0))
          tile[threadIdx.y + 1][threadIdx.x] = T[i][j - 1]; // left
        if (threadIdx.y == (0))
65        tile[threadIdx.y][threadIdx.x + 1] = T[i - 1][j]; // up
      }
      __syncthreads();

      if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1))) {
70      m = threadIdx.x + 1;
        l = threadIdx.y + 1;
        Txx = (tile[l][m + 1] - 2.0 * tile[l][m] + tile[l][m - 1])
            / dx2;
        Tyy = (tile[l + 1][m] - 2.0 * tile[l][m] + tile[l - 1][m])
            / dy2;
        tilen[l - 1][m - 1] = tile[l][m] + dt * C * (Txx + Tyy);
75    }
```

```
  if (j == 0)
    // tilen[threadIdx.y][threadIdx.x] = tile[threadIdx.y+1][
       threadIdx.x+1];
    tilen[threadIdx.y][threadIdx.x] = 0.0; // Condicion
       Dirchetl
  if (j == (nx - 1))
    // tilen[threadIdx.y][threadIdx.x] = tile[threadIdx.y+1][
       threadIdx.x+1];
    tilen[threadIdx.y][threadIdx.x] = 0.0;
  if (i == 0)
    // tilen[threadIdx.y][threadIdx.x] = tile[threadIdx.y+1][
       threadIdx.x+1];
    tilen[threadIdx.y][threadIdx.x] = 0.0;
  if (i == (ny - 1))
    // tilen[threadIdx.y][threadIdx.x] = tile[threadIdx.y+1][
       threadIdx.x+1];
    tilen[threadIdx.y][threadIdx.x] = 0.0;

  if ((i >= 0) && (i <= (ny - 1)) && (j >= 0) && (j <= (nx - 1)
     ))
    Tn[i][j] = tilen[threadIdx.y][threadIdx.x];
}

int main(int argc, char *argv[]) {
  real dx, dy;

#ifndef doble
  float C = 0.5f;
#else
  double C = 0.5;
#endif

  real dx2;
  real dy2;
  real dt;
  int nx, ny, i, j, t = 0;

  real **Tdev, **Tdev_n, **Temp2D;
  real *P_Tdev, *P_Tdev_n, *Temp;

  real **Thost;

  //----------------------------
  int numberofdevices, dev;
  cudaDeviceProp prop;
  // int blockSize;
  // int gridSize;
```

```
  dim3 grid, block;
  //----------------------------
  cudaEvent_t start, stop;
  float elapsedTime;

  int niter;
  int display;

  // si el bloque es de 8x8 luego entonces 64 threads!

  // nx = (int)(1.0f/dx)+1;
  // ny = (int)(1.0f/dy)+1;

  if (argc < 5) {

    printf("Falta el numero de iteraciones\n");
    exit(1);
  }

  printf("Numero de iteraciones: %s\n", argv[1]);
  printf("Numero de elementos en x: %s\n", argv[2]);
  printf("Numero de elementos en y: %s\n", argv[3]);
  printf("Muestreo: %s\n", argv[4]);

  niter = atoi(argv[1]);
  nx = atoi(argv[2]);
  ny = atoi(argv[3]);
  display = atoi(argv[4]);

  printf("---->Numero de iteraciones: %d\n", niter);
  printf("---->Numero de elementos en x: %d\n", nx);
  printf("---->Numero de elementos en y: %d\n", ny);
  printf("---->Display: %d", display);

  dx = 1.0 / ((real)nx - 1.0);
  dy = 1.0 / ((real)ny - 1.0);

  dx2 = dx * dx;
  dy2 = dy * dy;
  dt = (dx2 * dy2) / (2 * C * (dx2 + dy2));

  printf("\ndx:%f dy:%f dt:%f", dx, dy, dt);

  cudaSetDevice(2);
  cudaGetDeviceCount(&numberofdevices);
  cudaGetDevice(&dev);
  printf("\nNumber of devices: %d", numberofdevices);
```

```
  cudaGetDeviceCount(&numberofdevices);
  cudaSetDevice(2);
  cudaGetDevice(&dev);
  printf("\nNumber of devices: %d", numberofdevices);

  for (i = 0; i < numberofdevices; i++) {
    cudaGetDeviceProperties(&prop, i);
    printf("\n------- %d ----------", i);
    printf("\n\t Name: %s", prop.name);
    printf("\n\t Compute: %d.%d", prop.major, prop.minor);
    printf("\n\t Multiprocessor: %d", prop.multiProcessorCount)
        ;
    printf("\n\t Total global mem: %d \n", (long int)prop.
        totalGlobalMem);
  }
  printf("\nCurrent device:%d", dev);

  Thost = MATRIX_CPU(ny, nx, "Thost");
  MATRIX_GPU(&P_Tdev, &Tdev, ny, nx, "Tdev");
  MATRIX_GPU(&P_Tdev_n, &Tdev_n, ny, nx, "Tdev_n");

  for (i = 0; i < ny; i++) {
    for (j = 0; j < nx; j++) {
      Thost[i][j] = 0.0f;
    }
  }

  cudaMemcpy(P_Tdev, Thost[0], nx * ny * sizeof(real),
      cudaMemcpyHostToDevice);
  cudaMemcpy(P_Tdev_n, Thost[0], nx * ny * sizeof(real),
              cudaMemcpyHostToDevice);

  // Init for safe

  for (i = 0; i < ny; i++)
    for (j = 0; j < nx; j++) {

#ifndef doble

      if ((powf(((float)j + 1.0f) * dx - 0.5f, 2.0f) +
                powf(((float)i + 1.0f) * dy - 0.5f, 2.0f) <=
           0.1f) &&
          (powf(((float)j + 1.0f) * dx - 0.5f, 2.0f) +
                powf(((float)i + 1.0f) * dy - 0.5f, 2.0f) >=
           .05f))
        Thost[i][j] = 1.0f;
```

```
      else
        Thost[i][j] = 0.0f;

#else

      if ((pow(((double)j + 1.0) * dx - 0.5, 2.0) +
               pow(((double)i + 1.0) * dy - 0.5, 2.0) <=
           0.1) &&
          (pow(((double)j + 1.0) * dx - 0.5, 2.0) +
               pow(((double)i + 1.0) * dy - 0.5, 2.0f) >=
           .05)) {
        printf("%d  %d\n", i, j);
        Thost[i][j] = 1.0;
      } else
        Thost[i][j] = 0.0;

#endif
    }

  cudaMemcpy(P_Tdev, Thost[0], nx * ny * sizeof(real),
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU
  cudaMemcpy(P_Tdev_n, Thost[0], nx * ny * sizeof(real),
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU

  cudaMemcpy(Thost[0], P_Tdev, nx * ny * sizeof(real),
             cudaMemcpyDeviceToHost); // Transferir a la
                memoria del CPU

  write2D_in_time(&(Thost[0][0]), ny, nx, t); // regreso de la
     memoria....
  // writematrix(&(Thost[0]),ny,nx,t);

  block.x = tilex;
  block.y = tiley;

  printf("\nHasta aqui fueron las transferencias");
  getchar();

  grid.x = (int)ceil((real)(nx) / block.x);
  grid.y = (int)ceil((real)(ny) / block.y);

  printf("block.x: %d block.y: %d\n", block.x, block.y);
  printf("grid.x: %d grid.y: %d", grid.x, grid.y);
  getchar();

```

```
  // Medir el tiempo
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);
  //-----------------

  while (t < niter) {
#ifdef SHARED
    kernel_shared<<<grid, block>>>(Tdev, Tdev_n, ny, nx, dt, C,
        dx2, dy2);
#else
    kernel<<<grid, block>>>(Tdev, Tdev_n, ny, nx, dt, C, dx2,
       dy2);
#endif
    t++;

    if (t % display == 0) {
      printf("\nt:%d", t);
      cudaMemcpy(Thost[0], P_Tdev_n, nx * ny * sizeof(real),
                 cudaMemcpyDeviceToHost);
      cudaDeviceSynchronize();
      write2D_in_time(&(Thost[0][0]), ny, nx, t);
      // writematrix(&(Thost[0]),ny,nx,t);
    }

    Temp = P_Tdev;
    P_Tdev = P_Tdev_n;
    P_Tdev_n = Temp;

    Temp2D = Tdev;
    Tdev = Tdev_n;
    Tdev_n = Temp2D;
  }

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&elapsedTime, start, stop);

  printf("\nElapsed time: %f\n", elapsedTime);
  return 0;
}

void MATRIX_GPU(real **P, real ***M, int n, int m, char const *
   var) {

  int i;
  real **P_M, **dev_M;
```

```
295
  printf("\nVar:%s", var);

  P_M = (real **)malloc(n * sizeof(real *));

300 if (P_M == NULL) {

    printf("\nError Host Pointers");
    printf("\nP_M %p", P_M);
    exit(0);
305 }

  cudaMalloc((void **)&P_M[0], n * m * sizeof(real));

  if (P_M[0] == NULL) {
310   printf("\nError Host Pointers");
    printf("\nP_M %p", P_M[0]);
    exit(0);
  }

315 for (i = 1; i < n; i++)
    P_M[i] = P_M[0] + i * m;

  cudaMalloc((void ***)&dev_M, n * sizeof(real *));

320 if (dev_M == NULL) {
    printf("\nError dev Pointers");
    exit(0);
  }

325 cudaMemcpy(dev_M, P_M, n * sizeof(real *),
      cudaMemcpyHostToDevice);

  *(P) = P_M[0];
  *(M) = dev_M;
}
330
real **MATRIX_CPU(int n, int m, char const *var) {
  real **A = NULL;
  int i, j;

335 printf("\nAllocating matrix:  %s of total size %d", var, n *
      m);
  cudaMallocHost((void ***)&A, n * sizeof(real *));
  if (A == NULL)
    puts("Memory problem calloc first level"), exit(-1);
  cudaMallocHost((void **)&A[0], n * m * sizeof(real));
```

```
340   if (A[0] == NULL)
        puts("Memory second level"), exit(-1);
      for (i = 1; i < n; i++)
        A[i] = A[0] + i * m;
      for (i = 0; i < n; i++)
345     for (j = 0; j < m; j++)
          A[i][j] = 0.0;

      return A;
    }
350
    int write2D_in_time(real *M, int ny, int nx, int t) {
    #ifndef doble
      char dir[60] = "/home/ccouder/tmp/s-2D-2-2_";
    #else
355   char dir[60] = "/home/ccouder/tmp/2D-2-2_";
    #endif

      char num[6];
    #ifndef SHARED
360   char ext[] = ".dat";
    #else
      char ext[] = "S.dat";
    #endif
      int i, j;
365   FILE *f;

      sprintf(num, "%d", t);
      strcat(dir, num);
      strcat(dir, ext);
370
      printf("\nArchivo:%s\n", dir);

      f = fopen(dir, "w");
      for (i = 0; i < ny; i++) {
375     for (j = 0; j < nx; j++) {
    #ifndef doble
          fprintf(f, "%.8f\t", M[i * nx + j]);
    #else
          fprintf(f, "%.15lf\t", M[i * nx + j]);
380 #endif
        }
        fprintf(f, "\n");
      }
      fflush(f);
385   fclose(f);
      return 0;
```

```
}

void writematrix(real *start, int nx, int ny, int t) {
#ifndef doble
  char dir[60] = "/home/ccouder/tmp/s-2D-2-2_";
#else
  char dir[60] = "/home/ccouder/tmp/2D-2-2_";
#endif
  char num[6];
#ifndef SHARED
  char ext[] = ".bin";
#else
  char ext[] = "S.bin";
#endif

  // printf("\nDirectorio:%s\n", dir);
  sprintf(num, "%d", t);
  strcat(dir, num);
  strcat(dir, ext);

  printf("File :%s\n", dir);

  FILE *file = fopen(dir, "wb");

  fwrite(start, nx * ny * sizeof(real), 1, file);

  fclose(file);
}
```

A.4 KERNELS 3D

A.4.1 *Kernel 3D-1-1*

En el Listado 22 se muestra el código completo en CUDA C para el caso 2D, utilizando un arreglo unidimensional y un solo índice l. Se compila como `nvcc 3D-1-1.cu`, si se requiere el precisión doble se agrega la bandera `-Ddoble`, y para modificar el tamaño del `warp` utilizamos las banderas `Dtilex` y `Dtiley`, por ejemplo, si se requiere precisión doble con un `warp` de tamaño 64, se puede compilar como: `nvcc 3D-1-1.cu -Ddoble -Dtilex=8 -Dtiley=8`.

Listado 22: Código CUDA para el caso 3D, con un arreglo unidimensional y un índice.

```
#include <cuda.h>
```

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef doble
typedef float real;
#else
typedef double real;
#endif

#ifndef tilex
#define tilex 16
#endif

real **MATRIX_CPU(int n, int m, char const *var);
real ***CUBE_CPU(int n, int m, int z, char const *var);
int write3D_in_time(real *M, int ny, int nx, int nz, int t);

__global__ void kernel(real *T, real *Tn, int ny, int nx, int
   nz, real dt,
                       real C, real dx2, real dy2, real dz2) {
  int l;
  real Txx;
  real Tyy;
  real Tzz;
  l = blockIdx.x * blockDim.x + threadIdx.x;
  if (l > (nx * ny) && l < ((nx * ny) * (nz - 1)) &&
      !((l % nx) == 0 || ((l + 1) % nx == 0)) &&
      !((l % (nx * ny) < nx) || (((l - (nx * ny) + nx) % ((nx *
          ny)) < nx)))) {
    Txx = (T[l + 1] - 2.0 * T[l] + T[l - 1]) / dx2;
    Tyy = (T[l + nx] - 2.0 * T[l] + T[l - nx]) / dy2;
    Tzz = (T[l + nx * ny] - 2.0 * T[l] + T[l - nx * ny]) / dz2;
    Tn[l] = T[l] + dt * C * (Txx + Tyy + Tzz);
  }
}

int main(int argc, char *argv[]) {
  real dx, dy, dz;
  real C = 0.5;
  real dx2;
  real dy2;
  real dz2;

  real dt;
  int nx, ny, nz, i, j, k, t = 0;
```

```
  real *Tdev, *Thost, *Tdev_n, *Temp;
  //------------------------------
  int numberofdevices, dev;
  cudaDeviceProp prop;
  int blockSize;
  int gridSize;
  //------------------------------
  cudaEvent_t start, stop;
  float elapsedTime;

  //------------------------------
  int niter;
  int display;
  // int sizex;

  if (argc < 5) {
    printf("Faltan \n");
    exit(1);
  }

  printf("Numero de iteraciones: %s\n", argv[1]);
  printf("Numero de elementos en x: %s\n", argv[2]);
  printf("Numero de elementos en y: %s\n", argv[3]);
  printf("Numero de elementos en z: %s\n", argv[4]);
  printf("Muestreo: %s\n", argv[5]);

  niter = atoi(argv[1]);
  nx = atoi(argv[2]);
  ny = atoi(argv[3]);
  nz = atoi(argv[4]);
  display = atoi(argv[5]);

  printf("Numero de iteraciones: %d\n", niter);
  printf("Numero de elementos en x: %d\n", nx);
  printf("Numero de elementos en y: %d\n", ny);
  printf("Numero de elementos en z: %d\n", nz);
  printf("Muestreo: %d\n", display);

  dx = 1.0 / ((real)nx - 1.0);
  dy = 1.0 / ((real)ny - 1.0);
  dz = 1.0 / ((real)nz - 1.0);

  dx2 = dx * dx;
  dy2 = dy * dy;
  dz2 = dz * dz;
```

```
  dt = (dx2 * dy2 * dz2) / (2.0 * C * (dx2 * dy2 + dy2 * dz2 +
     dx2 * dz2));
95
  // nx = ceil(1.0f/dx)+1;
  // ny = ceil(1.0f/dy)+1;
  // nz = ceil(1.0f/dz)+1;

100  printf("\nnx:%f ny:%f nz:%f dt:%f", dx, dy, dz, dt);
  getchar();
  cudaSetDevice(2);
  cudaGetDeviceCount(&numberofdevices);
  cudaGetDevice(&dev);
105  printf("\nNumber of devices: %d", numberofdevices);

  for (i = 0; i < numberofdevices; i++) {
    cudaGetDeviceProperties(&prop, i);
    printf("\n------- %d ----------", i);
110    printf("\n\t Name: %s", prop.name);
    printf("\n\t Compute: %d.%d", prop.major, prop.minor);
    printf("\n\t Multiprocessor: %d", prop.multiProcessorCount)
       ;
    printf("\n\t Total global mem: %d \n", (long int)prop.
       totalGlobalMem);
  }
115  printf("\nDevice: %d", dev);
  cudaMallocHost((void **)&Thost, nx * ny * nz * sizeof(real));
  cudaMalloc((void **)&Tdev, nx * ny * nz * sizeof(real));
  cudaMalloc((void **)&Tdev_n, nx * ny * nz * sizeof(real));

120  cudaMemcpy(Tdev, Thost, nx * ny * nz * sizeof(real),
     cudaMemcpyHostToDevice);
  cudaMemcpy(Tdev_n, Thost, nx * ny * nz * sizeof(real),
             cudaMemcpyHostToDevice);

  for (i = 0; i < ny; i++)
125    for (j = 0; j < nx; j++)
      for (k = 0; k < nz; k++) {
#ifndef doble
        if ((powf(((float)j + 1.0) * dx - 0.5, 2.0) +
                 powf(((float)i + 1.0) * dy - 0.5, 2.0) +
130              powf(((float)k + 1.0) * dz - 0.5, 2.0) <=
            0.1) &&
           (powf(((float)j + 1.0) * dx - 0.5, 2.0) +
                 powf(((float)i + 1.0) * dy - 0.5, 2.0) +
                 powf(((float)k + 1.0) * dz - 0.5, 2.0) >=
135          .05))
          Thost[j + i * nx + k * nx * ny] = 1.0;
```

```
        else
          Thost[j + i * nx + k * nx * ny] = 0.0;

#else
        if ((pow(((double)j + 1.0) * dx - 0.5, 2.0) +
                 pow(((double)i + 1.0) * dy - 0.5, 2.0) +
                 pow(((double)k + 1.0) * dz - 0.5, 2.0) <=
            0.1) &&
           (pow(((double)j + 1.0) * dx - 0.5, 2.0) +
                 pow(((double)i + 1.0) * dy - 0.5, 2.0) +
                 pow(((double)k + 1.0) * dz - 0.5, 2.0) >=
            .05))
          Thost[j + i * nx + k * nx * ny] = 1.0;
        else
          Thost[j + i * nx + k * nx * ny] = 0.0;
#endif
      }

  cudaMemcpy(Tdev, Thost, nx * ny * nz * sizeof(real),
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU
  cudaMemcpy(Tdev_n, Thost, nx * ny * nz * sizeof(real),
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU

  cudaMemcpy(Thost, Tdev, nx * ny * nz * sizeof(real),
             cudaMemcpyDeviceToHost); // Transferir a la
                memoria del GPU
  write3D_in_time(Thost, ny, nx, nz, t);
  printf("\nWriting init Done");
  getchar();
  blockSize = tilex;
  gridSize = (int)ceil((real)(ny * nx * nz) / blockSize);
  printf("blockSize: %d, gridSize: %d", blockSize, gridSize);

  getchar();
  // Medir el tiempo
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);
  //-----------------
  while (t < niter) {
    kernel<<<gridSize, blockSize>>>(Tdev, Tdev_n, ny, nx, nz,
       dt, C, dx2, dy2,
                                    dz2);
    t++;
```

```
180     if (t % display == 0) {
          printf("\nt:%d", t);
          cudaMemcpy(Thost, Tdev_n, nx * ny * nz * sizeof(real),
                     cudaMemcpyDeviceToHost);
          cudaDeviceSynchronize();
185       write3D_in_time(&Thost[0], ny, nx, nz, t);
          // getchar();
        }

        Temp = Tdev;
190     Tdev = Tdev_n;
        Tdev_n = Temp;
      }
      cudaEventRecord(stop, 0);
      cudaEventSynchronize(stop);
195   cudaEventElapsedTime(&elapsedTime, start, stop);

      printf("\nElapsed time: %f", elapsedTime);
      return 0;
    }
200
    real ***CUBE_CPU(int n, int m, int z, char const *var) {
      real ***A = NULL;
      int i, j, k;

205   printf("\nAllocating a cube:  %s of total size %d", var, n *
          m * z);
      cudaMallocHost((void ****)&A, n * sizeof(real **));
      if (A == NULL)
        puts("Memory problem calloc first level"), exit(-1);
      cudaMallocHost((void ***)&A[0], n * m * sizeof(real *));
210
      for (i = 1; i < n; i++)
        A[i] = A[0] + i * m;

      cudaMallocHost((void **)&A[0][0], n * m * z * sizeof(real));
215
      for (j = 1; j < (n * m); j++)
        A[0][j] = A[0][0] + j * z;

      for (i = 0; i < n; i++)
220     for (j = 0; j < m; j++)
          for (k = 0; k < z; k++)
            A[i][j][k] = 0.0;

      return A;
225 }
```

```
void CUBE_GPU(real **P, real ****M, int n, int m, int z, char
    const *var) {

  int i, j;
  real ***P_M = NULL, ***dev_M = NULL;
  real **dev_M_2D = NULL;

  printf("\nSize : %d %d %d", n, m, z);
  printf("\nCUBE : %s", var);

  // fflush(stdout);

  printf("\nP_M: %p", P_M);

  cudaMallocHost((void ****)&P_M, n * sizeof(real **));
  // P_M = (real ***) calloc(n,sizeof(real**));

  if (P_M == NULL) {
    printf("\nError Host Pointers");
    printf("\nP_M %p", P_M);
    exit(0);
  }

  printf("\nP_M: %p", P_M);
  fflush(stdout);

  for (i = 0; i < n; i++)
    printf("\nP_M[%d]: %p, %p", i, P_M[i], &P_M[i]);

  // cudaMalloc( (void **)&P_M[0], n*m*sizeof(real));
  cudaMallocHost((void ***)&P_M[0], n * m * sizeof(real *));

  if (P_M[0] == NULL) {

    printf("\nError Host Pointers");
    printf("\nP_M %p", P_M[0]);
    exit(0);
  }

  for (i = 1; i < n; i++)
    P_M[i] = P_M[0] + i * m;

  cudaMalloc((void **)&P_M[0][0], n * m * z * sizeof(real));

  for (j = 1; j < (n * m); j++)
    P_M[0][j] = P_M[0][0] + j * z;
```

```
  cudaMalloc((void ****)&dev_M, n * sizeof(real **));
  cudaMalloc((void ***)&dev_M_2D, n * m * sizeof(real *));

  if (dev_M == NULL) {
    printf("\nError dev Pointers 3D");
    exit(0);
  }
  if (dev_M_2D == NULL) {
    printf("\nError dev Pointers 2D");
    exit(0);
  }

  printf("\ndev_M:%p", dev_M);
  printf("\ndev_M_2D:%p", dev_M_2D);

  cudaMemcpy(dev_M_2D, P_M[0], n * m * sizeof(real *),
     cudaMemcpyHostToDevice);

  *(P) = P_M[0][0];

  for (i = 0; i < n; i++)
    P_M[i] = dev_M_2D + i * m;

  cudaMemcpy(dev_M, P_M, n * sizeof(real **),
     cudaMemcpyHostToDevice);

  printf("\nCompletado..");
  fflush(stdout);
  *(M) = dev_M;

  printf("\nCompletado..");
  fflush(stdout);
}

int write3D_in_time(real *M, int ny, int nx, int nz, int t) {

#ifndef doble
  char dir[60] = "/home/ccouder/tmp/s-3D-1-1_";
#else
  char dir[60] = "/home/ccouder/tmp/3D-1-1_";
#endif

  char num[6];
  char ext[] = ".dat";
  int i, j, k;
  FILE *f;
```

```
  sprintf(num, "%d", t);
  strcat(dir, num);
  strcat(dir, ext);

  printf("File :%s", dir);
  f = fopen(dir, "w");

  for (i = 0; i < ny; i++)
    for (j = 0; j < nx; j++)
      for (k = 0; k < nz; k++) {
#ifndef doble
        fprintf(f, "%d %d %d %.8f\n", i + 1, j + 1, k + 1,
                M[j + i * nx + k * nx * ny]);
#else
        fprintf(f, "%d %d %d %.15f\n", i + 1, j + 1, k + 1,
                M[j + i * nx + k * nx * ny]);
#endif
      }

  fflush(f);
  fclose(f);
  return 0;
}
```

A.4.2 *Kernel 3D-1-3*

En el Listado 23 se muestra el código que contiene los `kernels` para el caso de un arreglo unidimensional y tres índices, con y sin el uso de memoria compartida. Las banderas de compilación son: `shared`, `double`, `tilex`, `tiley` y `tilez`. Por ejemplo, si deseamos utilizar memoria compartida, y crear un `warp` de `4 x 4 x 4`, y además utilizar precisión doble, deberemos compilar como: `nvcc -Dshared -Ddoble -Dtilex=4 -Dtiley=4 -Dtilez=4`.

Listado 23: Código CUDA para el caso 3D, con un arreglo unidimensional y tres índices.

```
#include <cuda.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef doble
typedef float real;
#else
```

```
typedef double real;
#endif

#ifndef tilex
#define tilex 4
#endif

#ifndef tiley
#define tiley 4
#endif

#ifndef tilez
#define tilez 4
#endif

real **MATRIX_CPU(int n, int m, char const *var);
real ***CUBE_CPU(int n, int m, int z, char const *var);
int write3D_in_time(real *M, int nx, int ny, int nz, int t);

__global__ void kernel(real *T, real *Tn, int ny, int nx, int
   nz, real dt,
                       real C, real dx2, real dy2, real dz2) {
  int i, j, k, l;
  j = blockIdx.x * blockDim.x + threadIdx.x;
  i = blockIdx.y * blockDim.y + threadIdx.y;
  k = blockIdx.z * blockDim.z + threadIdx.z;
  real Txx;
  real Tyy;
  real Tzz;

  if ((j > 0) && (j < (nx - 1)) && (i > 0) && (i < (ny - 1)) &&
      (k > 0) &&
      (k < (nz - 1))) {
    l = j + i * nx + k * (nx * ny);
    Txx = (T[l + 1] - 2.0 * T[l] + T[l - 1]) / dx2;
    Tyy = (T[l + nx] - 2.0 * T[l] + T[l - nx]) / dy2;
    Tzz = (T[l + nx * ny] - 2.0 * T[l] + T[l - nx * ny]) / dz2;
    Tn[l] = T[l] + dt * C * (Txx + Tyy + Tzz);
  }
}

__global__ void kernel_shared(real *T, real *Tn, int ny, int nx
   , int nz,
                              real dt, real C, real dx2, real
                                 dy2, real dz2) {

  int i, j, k, q, r, s, index;
```

```
  real Txx;
  real Tyy;
  real Tzz;

  __shared__ real tile[tiley + 2][tilex + 2][tilez + 2];
  __shared__ real tilen[tiley][tilex][tilez];

  j = blockIdx.x * blockDim.x + threadIdx.x;
  i = blockIdx.y * blockDim.y + threadIdx.y;
  k = blockIdx.z * blockDim.z + threadIdx.z;

  index = j + i * nx + k * (nx * ny);

  if ((i >= 0) && (i <= (ny - 1)) && (j >= 0) && (j <= (nx - 1)
     ) && (k >= 0) &&
      (k <= (nz - 1)))
    tile[threadIdx.y + 1][threadIdx.x + 1][threadIdx.z + 1] = T
       [index];

  if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1)) &&
      (k > 0) &&
      (k < (nz - 1))) {

    if (threadIdx.x == (0))
      tile[threadIdx.y + 1][threadIdx.x][threadIdx.z + 1] =
          T[index - 1]; // T[i][j-1][k]; //left

    if (threadIdx.x == (blockDim.x - 1))
      tile[threadIdx.y + 1][threadIdx.x + 2][threadIdx.z + 1] =
          T[index + 1]; // T[i][j+1][k]; //right

    if (threadIdx.y == (0))
      tile[threadIdx.y][threadIdx.x + 1][threadIdx.z + 1] =
          T[index - nx]; // T[i-1][j][k]; //up  */

    if (threadIdx.y == (blockDim.y - 1))
      tile[threadIdx.y + 2][threadIdx.x + 1][threadIdx.z + 1] =
          T[index + nx]; // T[i+1][j][k]; //down

    if (threadIdx.z == (0))
      tile[threadIdx.y + 1][threadIdx.x + 1][threadIdx.z] =
          T[index - nx * ny]; // T[i][j][k-1];

    if (threadIdx.z == (blockDim.z - 1))
      tile[threadIdx.y + 1][threadIdx.x + 1][threadIdx.z + 2] =
          T[index + nx * ny]; // T[i][j][k+1];
  }
```

```
  __syncthreads();

  if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1)) &&
      (k > 0) &&
      (k < (nz - 1))) {
    q = threadIdx.x + 1;
    r = threadIdx.y + 1;
    s = threadIdx.z + 1;

    Txx = (tile[r][q + 1][s] - 2.0 * tile[r][q][s] + tile[r][q
       - 1][s]) / dx2;
    Tyy = (tile[r + 1][q][s] - 2.0 * tile[r][q][s] + tile[r -
       1][q][s]) / dy2;
    Tzz = (tile[r][q][s + 1] - 2.0 * tile[r][q][s] + tile[r][q
       ][s - 1]) / dz2;

    tilen[r - 1][q - 1][s - 1] = tile[r][q][s] + dt * C * (Txx
       + Tyy + Tzz);
  }

  if (j == 0)
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0; //
       Condicion Dirchetl

  if (j == (nx - 1))
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;

  if (i == 0)
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;

  if (i == (ny - 1))
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;

  if (k == 0)
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;

  if (k == (nz - 1))
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;

  if ((i >= 0) && (i <= (ny - 1)) && (j >= 0) && (j <= (nx - 1)
     ) && (k >= 0) &&
      (k <= (nz - 1)))
    Tn[index] = tilen[threadIdx.y][threadIdx.x][threadIdx.z];
}

int main(int argc, char *argv[]) {
```

```
  real dx, dy, dz;
  real C = 0.5;
  real dx2;
  real dy2;
  real dz2;

  real dt;
  int nx, ny, nz, i, j, k, t = 0;
  real *Tdev, *Thost, *Tdev_n, *Temp;
  //-----------------------------
  int numberofdevices, dev;
  cudaDeviceProp prop;
  dim3 block;
  dim3 grid;
  //-----------------------------
  cudaEvent_t start, stop;
  float elapsedTime;

  int niter;
  int display;

  if (argc < 6) {
    printf("Faltan los parametros \n");
    exit(1);
  }

  printf("Numero de iteraciones: %s\n", argv[1]);
  printf("Numero de elementos en x: %s\n", argv[2]);
  printf("Numero de elementos en y: %s\n", argv[3]);
  printf("Numero de elementos en z: %s\n", argv[4]);
  printf("Muestreo: %s\n", argv[5]);

  niter = atoi(argv[1]);
  nx = atoi(argv[2]);
  ny = atoi(argv[3]);
  nz = atoi(argv[4]);
  display = atoi(argv[5]);

  printf("Numero de iteraciones: %d\n", niter);
  printf("Numero de elementos en x: %d\n", nx);
  printf("Numero de elementos en y: %d\n", ny);
  printf("Numero de elementos en z: %d\n", nz);
  printf("Muestreo: %d\n", display);

  block.x = tilex; // sizex
  block.y = tiley; // sizey
  block.z = tilez; // sizey
```

```
185   // nx = ceil(1.0f/dx)+1;
      // ny = ceil(1.0f/dy)+1;
      // nz = ceil(1.0f/dz)+1;

      dx = 1.0 / ((real)nx - 1.0);
190   dy = 1.0 / ((real)ny - 1.0);
      dz = 1.0 / ((real)nz - 1.0);

      dx2 = dx * dx;
      dy2 = dy * dy;
195   dz2 = dz * dz;

      dt = (dx2 * dy2 * dz2) / (2.0 * C * (dx2 * dy2 + dy2 * dz2 +
          dx2 * dz2));

      printf("\nnx:%f ny:%f nz:%f dt:%f", dx, dy, dz, dt);
200   getchar();
      cudaSetDevice(2);
      cudaGetDeviceCount(&numberofdevices);
      cudaGetDevice(&dev);
      printf("\nNumber of devices: %d", numberofdevices);
205
      for (i = 0; i < numberofdevices; i++) {
        cudaGetDeviceProperties(&prop, i);
        printf("\n------- %d ----------", i);
        printf("\n\t Name: %s", prop.name);
210     printf("\n\t Compute: %d.%d", prop.major, prop.minor);
        printf("\n\t Multiprocessor: %d", prop.multiProcessorCount)
            ;
        printf("\n\t Total global mem: %d \n", (long int)prop.
            totalGlobalMem);
      }
      cudaMallocHost((void **)&Thost, nx * ny * nz * sizeof(real));
215   cudaMalloc((void **)&Tdev, nx * ny * nz * sizeof(real));
      cudaMalloc((void **)&Tdev_n, nx * ny * nz * sizeof(real));

      printf("\nDevice: %d", dev);

220   cudaMemcpy(Tdev, Thost, nx * ny * nz * sizeof(real),
          cudaMemcpyHostToDevice);
      cudaMemcpy(Tdev_n, Thost, nx * ny * nz * sizeof(real),
                 cudaMemcpyHostToDevice);

      for (i = 0; i < ny; i++)
225     for (j = 0; j < nx; j++)
          for (k = 0; k < nz; k++) {
```

```
#ifndef doble
        if ((powf(((float)j + 1.0) * dx - 0.5, 2.0) +
                  powf(((float)i + 1.0) * dy - 0.5, 2.0) +
                  powf(((float)k + 1.0) * dz - 0.5, 2.0) <=
             0.1) &&
            (powf(((float)j + 1.0) * dx - 0.5, 2.0) +
                  powf(((float)i + 1.0) * dy - 0.5, 2.0) +
                  powf(((float)k + 1.0) * dz - 0.5, 2.0) >=
             .05))
          Thost[j + i * nx + k * nx * ny] = 1.0;
        else
          Thost[j + i * nx + k * nx * ny] = 0.0;

#else
        if ((pow(((double)j + 1.0) * dx - 0.5, 2.0) +
                  pow(((double)i + 1.0f) * dy - 0.5, 2.0) +
                  pow(((double)k + 1.0) * dz - 0.5, 2.0) <=
             0.1) &&
            (pow(((double)j + 1.0) * dx - 0.5, 2.0) +
                  pow(((double)i + 1.0) * dy - 0.5, 2.0) +
                  pow(((double)k + 1.0) * dz - 0.5, 2.0) >=
             .05))
          Thost[j + i * nx + k * nx * ny] = 1.0;
        else
          Thost[j + i * nx + k * nx * ny] = 0.0;
#endif
      }

  cudaMemcpy(Tdev, Thost, nx * ny * nz * sizeof(real),
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU
  cudaMemcpy(Tdev_n, Thost, nx * ny * nz * sizeof(real),
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU

  cudaMemcpy(Thost, Tdev, nx * ny * nz * sizeof(real),
             cudaMemcpyDeviceToHost); // Transferir a la
                memoria del GPU
  write3D_in_time(Thost, ny, nx, nz, t);
  printf("\nWriting init");
  getchar();

  grid.x = (int)ceil((real)(nx) / block.x);
  grid.y = (int)ceil((real)(ny) / block.y);
  grid.z = (int)ceil((real)(nz) / block.z);
```

```
270   printf("block.x: %d block.y: %d  block.z: %d\n", block.x,
          block.y, block.z);
      printf("grid.x: %d grid.y: %d  grid.z: %d", grid.x, grid.y,
          grid.z);
      getchar();

      // Medir el tiempo
275   cudaEventCreate(&start);
      cudaEventCreate(&stop);
      cudaEventRecord(start, 0);
      //-----------------

280   while (t < niter) {
        kernel<<<grid, block>>>(Tdev, Tdev_n, ny, nx, nz, dt, C,
            dx2, dy2, dz2);
        t++;

        if (t % display == 0) {
285       printf("\nt:%d", t);
          cudaMemcpy(Thost, Tdev_n, nx * ny * nz * sizeof(real),
                     cudaMemcpyDeviceToHost);
          cudaDeviceSynchronize();
          write3D_in_time(Thost, ny, nx, nz, t);
290     }
        Temp = Tdev;
        Tdev = Tdev_n;
        Tdev_n = Temp;
      }
295
      cudaEventRecord(stop, 0);
      cudaEventSynchronize(stop);
      // CALCULATE ELAPSED TIME
      cudaEventElapsedTime(&elapsedTime, start, stop);
300
      printf("\nElapsed time: %f\n", elapsedTime);

      return 0;
    }
305
    real ***CUBE_CPU(int n, int m, int z, char const *var) {
      real ***A = NULL;
      // float * p = NULL;

310   int i, j, k;

      printf("\nAllocating a cube:  %s of total size %d", var, n *
          m * z);
```

```
  cudaMallocHost((void ****)&A, n * sizeof(real **));
  if (A == NULL)
    puts("Memory problem calloc first level"), exit(-1);

  cudaMallocHost((void ***)&A[0], n * m * sizeof(real *));

  for (i = 1; i < n; i++)
    A[i] = A[0] + i * m;

  cudaMallocHost((void **)&A[0][0], n * m * z * sizeof(real));

  for (j = 1; j < (n * m); j++)
    A[0][j] = A[0][0] + j * z;

  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      for (k = 0; k < z; k++) {
        A[i][j][k] = 0.0;
      }

  return A;
}

void CUBE_GPU(real **P, real ****M, int n, int m, int z, char
    const *var) {

  int i, j;
  real ***P_M = NULL, ***dev_M = NULL;
  real **dev_M_2D = NULL;

  printf("\nSize : %d %d %d", n, m, z);
  printf("\nCUBE : %s", var);

  // fflush(stdout);

  printf("\nP_M: %p", P_M);

  cudaMallocHost((void ****)&P_M, n * sizeof(real **));
  // P_M = (real ***) calloc(n,sizeof(real**));

  if (P_M == NULL) {
    printf("\nError Host Pointers");
    printf("\nP_M %p", P_M);
    exit(0);
  }
```

```
  printf("\nP_M: %p", P_M);
  fflush(stdout);

  for (i = 0; i < n; i++)
    printf("\nP_M[%d]: %p, %p", i, P_M[i], &P_M[i]);

  // cudaMalloc( (void **)&P_M[0], n*m*sizeof(real));
  cudaMallocHost((void ***)&P_M[0], n * m * sizeof(real *));

  if (P_M[0] == NULL) {

    printf("\nError Host Pointers");
    printf("\nP_M %p", P_M[0]);
    exit(0);
  }

  for (i = 1; i < n; i++)
    P_M[i] = P_M[0] + i * m;

  cudaMalloc((void **)&P_M[0][0], n * m * z * sizeof(real));

  // cudaMalloc((void**)&P_M[0][0], n*m*z*sizeof(real));

  for (j = 1; j < (n * m); j++)
    P_M[0][j] = P_M[0][0] + j * z;

  cudaMalloc((void ****)&dev_M, n * sizeof(real **));
  cudaMalloc((void ***)&dev_M_2D, n * m * sizeof(real *));

  if (dev_M == NULL) {
    printf("\nError dev Pointers 3D");
    exit(0);
  }
  if (dev_M_2D == NULL) {
    printf("\nError dev Pointers 2D");
    exit(0);
  }

  printf("\ndev_M:%p", dev_M);
  printf("\ndev_M_2D:%p", dev_M_2D);

  cudaMemcpy(dev_M_2D, P_M[0], n * m * sizeof(real *),
     cudaMemcpyHostToDevice);

  printf("\nCopiado 2D completado ");
  // fflush(stdout);
  printf("\nTransferencia al puntero");
```

```
405   *(P) = P_M[0][0];

      for (i = 0; i < n; i++)
        P_M[i] = dev_M_2D + i * m;

410   cudaMemcpy(dev_M, P_M, n * sizeof(real **),
         cudaMemcpyHostToDevice);

      printf("\nCompletado..");
      fflush(stdout);
      *(M) = dev_M;
415
      printf("\nCompletado..");
      fflush(stdout);
      // getchar();
}
420
int write3D_in_time(real *M, int ny, int nx, int nz, int t) {

#ifndef doble
      char dir[60] = "/home/ccouder/tmp/s-3D-1-3_";
425 #else
      char dir[60] = "/home/ccouder/tmp/3D-1-3_";
#endif

      char num[6];
430
#ifndef SHARED
      char ext[] = ".dat";
#else
      char ext[] = "S.dat";
435 #endif

      int i, j, k;
      FILE *f;

440   sprintf(num, "%d", t);
      strcat(dir, num);
      strcat(dir, ext);

      printf("File :%s", dir);
445   f = fopen(dir, "w");

      for (i = 0; i < ny; i++)
        for (j = 0; j < nx; j++)
          for (k = 0; k < nz; k++) {
450
```

```
#ifndef doble
        fprintf(f, "%d %d %d %.8f\n", i + 1, j + 1, k + 1,
                M[j + i * nx + k * nx * ny]);
#else
        fprintf(f, "%d %d %d %.15lf\n", i + 1, j + 1, k + 1,
                M[j + i * nx + k * nx * ny]);
#endif
      }

  fflush(f);
  fclose(f);
  return 0;}
```

A.4.3 *Kernel 3D-3-3*

En el Listado 24 se muestra el código que contiene los `kernels` para el caso de un arreglo tridimensional y tres índices. Las banderas de compilación son: `shared`, `double`, `tilex`, `tiley` y `tilez`. Por ejemplo, si deseamos utilizar memoria compartida, y crear un `warp` de `8 x 8`, y además utilizar precisión doble, deberemos compilar como: `nvcc -Dshared -Ddoble -Dtilex=8 -Dtiley=8`.

Listado 24: Código CUDA para el caso 2D, con un arreglo bidimensional y dos índices.

```
#include <cuda.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef doble
typedef float real;
#else
typedef double real;
#endif

#ifndef tilex
#define tilex 4
#endif

#ifndef tiley
#define tiley 4
#endif

#ifndef tilez
#define tilez 4
```

```
#endif

void CUBE_GPU(real **P, real ****M, int n, int m, int z, char
    const *var);
real ***CUBE_CPU(int n, int m, int z, char const *var);
int write3D_in_time(real ***M, int ny, int nx, int nz, int t);

__global__ void kernel(real ***T, real ***Tn, int ny, int nx,
    int nz, real dt,
                       real C, real dx2, real dy2, real dz2) {
  int i, j, k;
  j = blockIdx.x * blockDim.x + threadIdx.x;
  i = blockIdx.y * blockDim.y + threadIdx.y;
  k = blockIdx.z * blockDim.z + threadIdx.z;
  real Txx;
  real Tyy;
  real Tzz;
  if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1)) &&
      (k > 0) &&
      (k < (nz - 1))) {
    Tzz = (T[i][j][k + 1] - 2.0 * T[i][j][k] + T[i][j][k - 1])
        / dz2;
    Txx = (T[i][j + 1][k] - 2.0 * T[i][j][k] + T[i][j - 1][k])
        / dx2;
    Tyy = (T[i + 1][j][k] - 2.0 * T[i][j][k] + T[i - 1][j][k])
        / dy2;

    Tn[i][j][k] = T[i][j][k] + dt * C * (Txx + Tyy + Tzz);
  }
}

__global__ void kernel_shared(real ***T, real ***Tn, int ny,
    int nx, int nz,
                              real dt, real C, real dx2, real
                                 dy2, real dz2) {
  int i, j, k, q, r, s;
  real Txx;
  real Tyy;
  real Tzz;

  __shared__ real tile[tiley + 2][tilex + 2][tilez + 2];
  __shared__ real tilen[tiley][tilex][tilez];

  j = blockIdx.x * blockDim.x + threadIdx.x;
  i = blockIdx.y * blockDim.y + threadIdx.y;
  k = blockIdx.z * blockDim.z + threadIdx.z;
```

```
  if ((i >= 0) && (i <= (ny - 1)) && (j >= 0) && (j <= (nx - 1)
      ) && (k >= 0) &&
       (k <= (nz - 1)))
    tile[threadIdx.y + 1][threadIdx.x + 1][threadIdx.z + 1] = T
       [i][j][k];

  if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1)) &&
       (k > 0) &&
       (k < (nz - 1))) {

    if (threadIdx.x == (0))
      tile[threadIdx.y + 1][threadIdx.x][threadIdx.z + 1] =
          T[i][j - 1][k]; // left

    if (threadIdx.x == (blockDim.x - 1))
      tile[threadIdx.y + 1][threadIdx.x + 2][threadIdx.z + 1] =
          T[i][j + 1][k]; // right

    if (threadIdx.y == (0))
      tile[threadIdx.y][threadIdx.x + 1][threadIdx.z + 1] =
          T[i - 1][j][k]; // up */

    if (threadIdx.y == (blockDim.y - 1))
      tile[threadIdx.y + 2][threadIdx.x + 1][threadIdx.z + 1] =
          T[i + 1][j][k]; // down

    if (threadIdx.z == (0))
      tile[threadIdx.y + 1][threadIdx.x + 1][threadIdx.z] = T[i
         ][j][k - 1];

    if (threadIdx.z == (blockDim.z - 1))
      tile[threadIdx.y + 1][threadIdx.x + 1][threadIdx.z + 2] =
          T[i][j][k + 1];
  }

  __syncthreads();

  if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1)) &&
       (k > 0) &&
       (k < (nz - 1))) {
    q = threadIdx.x + 1;
    r = threadIdx.y + 1;
    s = threadIdx.z + 1;

    Txx = (tile[r][q + 1][s] - 2.0 * tile[r][q][s] + tile[r][q
       - 1][s]) / dx2;
```

```
    Tyy = (tile[r + 1][q][s] - 2.0 * tile[r][q][s] + tile[r -
        1][q][s]) / dy2;
    Tzz = (tile[r][q][s + 1] - 2.0 * tile[r][q][s] + tile[r][q
        ][s - 1]) / dz2;

    tilen[r - 1][q - 1][s - 1] = tile[r][q][s] + dt * C * (Txx
        + Tyy + Tzz);
  }

  if (j == 0)
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0; //
        Condicion Dirchetl

  if (j == (nx - 1))
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;

  if (i == 0)
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;

  if (i == (ny - 1))
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;

  if (k == 0)
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;

  if (k == (nz - 1))
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;

  if ((i >= 0) && (i <= (ny - 1)) && (j >= 0) && (j <= (nx - 1)
      ) && (k >= 0) &&
       (k <= (nz - 1)))
    Tn[i][j][k] = tilen[threadIdx.y][threadIdx.x][threadIdx.z];
}

int main(int argc, char *argv[]) {
  real dx, dy, dz;
  real C = 0.5;
  real dx2;
  real dy2;
  real dz2;

  real dt;
  int nx, ny, nz, i, j, k, t = 0;
  real ***Tdev, ***Tdev_n, ***Temp3D;
  real ***Thost;
  real *P_Tdev, *P_Tdev_n, *Temp;
  //----------------------------
```

```
  int numberofdevices, dev;
  cudaDeviceProp prop;
145 dim3 block;
  dim3 grid;
  //----------------------------

  cudaEvent_t start, stop;
150 float elapsedTime;

  int niter;
  int display;

155 if (argc < 6) {
    printf("Faltan los parametros \n");
    exit(1);
  }

160 printf("Numero de iteraciones: %s\n", argv[1]);
  printf("Numero de elementos en x: %s\n", argv[2]);
  printf("Numero de elementos en y: %s\n", argv[3]);
  printf("Numero de elementos en z: %s\n", argv[4]);
  printf("Muestreo: %s\n", argv[5]);
165
  niter = atoi(argv[1]);
  nx = atoi(argv[2]);
  ny = atoi(argv[3]);
  nz = atoi(argv[4]);
170 display = atoi(argv[5]);

  printf("Numero de iteraciones: %d\n", niter);
  printf("Numero de elementos en x: %d\n", nx);
  printf("Numero de elementos en y: %d\n", ny);
175 printf("Numero de elementos en z: %d\n", nz);
  printf("Muestreo: %d\n", display);

  block.x = tilex; // sizex
  block.y = tiley; // sizey
180 block.z = tilez; // sizey

  dx = 1.0 / ((real)nx - 1.0);
  dy = 1.0 / ((real)ny - 1.0);
  dz = 1.0 / ((real)nz - 1.0);
185
  dx2 = dx * dx;
  dy2 = dy * dy;
  dz2 = dz * dz;
```

```
190   dt = (dx2 * dy2 * dz2) / (2.0 * C * (dx2 * dy2 + dy2 * dz2 +
          dx2 * dz2));

      printf("\nnx:%f ny:%f nz:%f dt:%f", dx, dy, dz, dt);
      getchar();
      cudaSetDevice(2);
195   cudaGetDeviceCount(&numberofdevices);
      cudaGetDevice(&dev);
      printf("\nNumber of devices: %d", numberofdevices);
      printf("\nDevice: %d", dev);

200   for (i = 0; i < numberofdevices; i++) {
        cudaGetDeviceProperties(&prop, i);
        printf("\n------- %d ----------", i);
        printf("\n\t Name: %s", prop.name);
        printf("\n\t Compute: %d.%d", prop.major, prop.minor);
205     printf("\n\t Multiprocessor: %d", prop.multiProcessorCount)
            ;
        printf("\n\t Total global mem: %d \n", (long int)prop.
            totalGlobalMem);
      }

      Thost = CUBE_CPU(ny, nx, nz, "Thost");
210   CUBE_GPU(&P_Tdev, &Tdev, ny, nx, nz, "Tdev");
      CUBE_GPU(&P_Tdev_n, &Tdev_n, ny, nx, nz, "Tdev_n");

      // Init for safe
      for (i = 0; i < ny; i++)
215     for (j = 0; j < nx; j++)
          for (k = 0; k < nz; k++)
            Thost[i][j][k] = 0.0;

      cudaMemcpy(P_Tdev, Thost[0][0], nx * ny * nz * sizeof(real),
220               cudaMemcpyHostToDevice);
      cudaMemcpy(P_Tdev_n, Thost[0][0], nx * ny * nz * sizeof(real)
          ,
                cudaMemcpyHostToDevice);

      for (i = 0; i < ny; i++)
225     for (j = 0; j < nx; j++)
          for (k = 0; k < nz; k++) {

    #ifndef doble
            if ((powf(((float)j + 1.0) * dx - 0.5, 2.0) +
230               powf(((float)i + 1.0) * dy - 0.5, 2.0) +
                  powf(((float)k + 1.0) * dz - 0.5, 2.0) <=
                0.1) &&
```

```
            (powf(((float)j + 1.0) * dx - 0.5, 2.0) +
                 powf(((float)i + 1.0) * dy - 0.5, 2.0) +
                 powf(((float)k + 1.0) * dz - 0.5, 2.0) >=
             .05))
          Thost[i][j][k] = 1.0;
        else
          Thost[i][j][k] = 0.0;

#else
        if ((pow(((double)j + 1.0) * dx - 0.5, 2.0) +
                 pow(((double)i + 1.0) * dy - 0.5, 2.0) +
                 pow(((double)k + 1.0) * dz - 0.5, 2.0) <=
             0.1) &&
            (pow(((double)j + 1.0) * dx - 0.5, 2.0) +
                 pow(((double)i + 1.0) * dy - 0.5, 2.0) +
                 pow(((double)k + 1.0) * dz - 0.5, 2.0) >=
             .05))
          Thost[i][j][k] = 1.0;
        else
          Thost[i][j][k] = 0.0;
#endif
      }
  // write3D(T, ny, nx, nz);

  cudaMemcpy(P_Tdev, Thost[0][0], nx * ny * nz * sizeof(real),
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU
  cudaMemcpy(P_Tdev_n, Thost[0][0], nx * ny * nz * sizeof(real)
     ,
             cudaMemcpyHostToDevice); // Transferir a la
                memoria del GPU

  cudaMemcpy(Thost[0][0], P_Tdev, nx * ny * nz * sizeof(real),
             cudaMemcpyDeviceToHost); // Transferir a la
                memoria del GPU
  grid.x = (int)ceil((real)(nx) / block.x);
  grid.y = (int)ceil((real)(ny) / block.y);
  grid.z = (int)ceil((real)(nz) / block.z);

  printf("block.x: %d block.y: %d  block.z: %d\n", block.x,
     block.y, block.z);
  printf("grid.x: %d grid.y: %d  grid.z: %d", grid.x, grid.y,
     grid.z);
  getchar();

  write3D_in_time(Thost, ny, nx, nz, t);
```

```
  cudaDeviceSynchronize();
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);
  //------------------
  while (t < niter) {
#ifdef SHARED
    kernel_shared<<<grid, block>>>(Tdev, Tdev_n, ny, nx, nz, dt
       , C, dx2, dy2,
                                   dz2);
#else
    kernel<<<grid, block>>>(Tdev, Tdev_n, ny, nx, nz, dt, C,
       dx2, dy2, dz2);
#endif

    t++;

    if (t % display == 0) {
      printf("\nt:%d", t);
      cudaMemcpy(Thost[0][0], P_Tdev_n, nx * ny * nz * sizeof(
         real),
                 cudaMemcpyDeviceToHost);
      cudaDeviceSynchronize();
      write3D_in_time(Thost, ny, nx, nz, t);
    }

    Temp = P_Tdev;
    P_Tdev = P_Tdev_n;
    P_Tdev_n = Temp;

    Temp3D = Tdev;
    Tdev = Tdev_n;
    Tdev_n = Temp3D;
  }

  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);
  cudaEventElapsedTime(&elapsedTime, start, stop);

  printf("\nElapsed time: %f\n", elapsedTime);

  return 0;
}

real ***CUBE_CPU(int n, int m, int z, char const *var) {
  real ***A = NULL;
```

```
  int i, j, k;

  printf("\nAllocating a cube:  %s of total size %d", var, n *
      m * z);

  cudaMallocHost((void ****)&A, n * sizeof(real **));
  if (A == NULL)
    puts("Memory problem calloc first level"), exit(-1);

  cudaMallocHost((void ***)&A[0], n * m * sizeof(real *));

  for (i = 1; i < n; i++)
    A[i] = A[0] + i * m;

  cudaMallocHost((void **)&A[0][0], n * m * z * sizeof(real));

  for (j = 1; j < (n * m); j++)
    A[0][j] = A[0][0] + j * z;

  for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      for (k = 0; k < z; k++) {
        A[i][j][k] = 0.0;
      }

  return A;
}

void CUBE_GPU(real **P, real ****M, int n, int m, int z, char
    const *var) {

  int i, j;
  real ***P_M = NULL, ***dev_M = NULL;
  real **dev_M_2D = NULL;

  printf("\nSize : %d %d %d", n, m, z);
  printf("\nCUBE : %s", var);

  printf("\nP_M: %p", P_M);

  cudaMallocHost((void ****)&P_M, n * sizeof(real **));

  if (P_M == NULL) {
    printf("\nError Host Pointers");
    printf("\nP_M %p", P_M);
    exit(0);
  }
```

```
  cudaMallocHost((void ***)&P_M[0], n * m * sizeof(real *));

  if (P_M[0] == NULL) {
    printf("\nError Host Pointers");
    printf("\nP_M %p", P_M[0]);
    exit(0);
  }

  for (i = 1; i < n; i++)
    P_M[i] = P_M[0] + i * m;

  cudaMalloc((void **)&P_M[0][0], n * m * z * sizeof(real));

  for (j = 1; j < (n * m); j++)
    P_M[0][j] = P_M[0][0] + j * z;

  cudaMalloc((void ****)&dev_M, n * sizeof(real **));
  cudaMalloc((void ***)&dev_M_2D, n * m * sizeof(real *));

  if (dev_M == NULL) {
    printf("\nError dev Pointers 3D");
    exit(0);
  }
  if (dev_M_2D == NULL) {
    printf("\nError dev Pointers 2D");
    exit(0);
  }

  printf("\ndev_M:%p", dev_M);
  printf("\ndev_M_2D:%p", dev_M_2D);

  cudaMemcpy(dev_M_2D, P_M[0], n * m * sizeof(real *),
     cudaMemcpyHostToDevice);

  printf("\nCopiado 2D completado ");
  // fflush(stdout);
  printf("\nTransferencia al puntero");
  *(P) = P_M[0][0];

  for (i = 0; i < n; i++)
    P_M[i] = dev_M_2D + i * m;

  cudaMemcpy(dev_M, P_M, n * sizeof(real **),
     cudaMemcpyHostToDevice);

  printf("\nCompletado..");
```

```
  fflush(stdout);
  *(M) = dev_M;

  printf("\nCompletado..");
  fflush(stdout);
  // getchar();
}

int write3D_in_time(real ***M, int ny, int nx, int nz, int t) {

#ifndef doble
  char dir[60] = "/home/ccouder/tmp/s-3D-3-3_";
#else
  char dir[60] = "/home/ccouder/tmp/3D-3-3_";
#endif

  char num[6];

#ifndef SHARED
  char ext[] = ".dat";
#else
  char ext[] = "S.dat";
#endif
  int i, j, k;
  FILE *f;

  sprintf(num, "%d", t);
  strcat(dir, num);
  strcat(dir, ext);
  printf("File :%s", dir);
  f = fopen(dir, "w");

  for (i = 0; i < ny; i++)
    for (j = 0; j < nx; j++)
      for (k = 0; k < nz; k++) {
#ifndef doble
        fprintf(f, "%d %d %d %.8f\n", i + 1, j + 1, k + 1, M[i
           ][j][k]);
#else
        fprintf(f, "%d %d %d %.15lf\n", i + 1, j + 1, k + 1, M[
           i][j][k]);
#endif
      }

  fflush(f);
  fclose(f);
  return 0;
```

```
}
```

B

PRESENTACIONES EN CONGRESOS

Este trabajo fue presentado en el *The Latin America High Performance Computing Conference comes to Mexico*(Ver Figura 2.29).

Figura 2.29: Cartel de presentación del congreso.

A pesar de que el trabajo no es precisamente de super-cómputo, ya que no aborda el uso de computadoras de alto rendimiento, por su interés y calidad se incluirá en la revista *Lecture Notes in Computer Science* https://www.springer.com/series/558 [1].

A continuación se incluye el PDF del *Proceeding* enviado.

1 Then, as we inform you before, your work is recommended to be present in the Advanced Computing Trends Workshop. However, given the high quality of your work the Program Committee has decided to consider your work for the LNCC springer publication, #CARLA2021 Committee

Solving the heat transfer equation by a finite difference method using multi-dimensional arrays in CUDA as in standard C.*

Josefina Sanchez-Noguez[1], Carlos Couder-Castañeda[2][0000−0001−8826−3750], J. J. Hernández-Gómez[2][0000−0002−5012−6619], and Itzel Navarro-Reyes[3]

[1] Facultad de Estudios Superiores Acatlán, UNAM, Mexico
[2] Centro de Desarrollo Aeroespacial, Insituto Politécnico Nacional, Mexico
ccouder@ipn.mx
[3] Escuela Superior de Física y Matemáticas, Instituto Politécnico Nacional, Mexico

Abstract. In recent years the increasing necessity to speed up the execution of numerical algorithms has leaded researchers to the use of coprocessors and graphic cards such as the NVIDIA GPU's. Despite CUDA C meta-language was introduced to facilitate the development of general purpose-applications, the solution to the common question: How to allocate (cudaMalloc) two-dimensional array?, is not simple. In this paper, we present a memory structure that allows the use of multidimensional arrays inside a CUDA kernel, to demonstrate its functionality, this structure is applied to the explicit finite difference solution of the non-steady heat transport equation.

Keywords: CUDA C · Multiarrays · Heat Transfer.

1 Introduction

Today, the use of Graphic processing units (GPUs) to accelerate scientific applications is very common alongside the multi-core architectures to speed up a huge range of engineering applications to reduce their computing time. NVIDIA GPUs initially used to improve the performance of video games were not accessible to develop general purpose applications; nevertheless, with the introduction of CUDA (Compute Unified Device Architecture), the power of GPUs for scientific and engineering applications was unleashed by extensions to C language [12].

For this reason, GPUs have become part of the most powerful supercomputers on earth (see `http://www.top500.org`). Notwithstanding this remarkable fact, the usage of GPUs in scientific computation using low-level tools as CUDA C is a complex task because it requires a high knowledge of the GPU architecture to obtain the best speed-up benefits.

Nowadays, hundreds of scientific applications have been migrated to GPU, so it is impossible to mention all of them, but the most current and representative

* This work was partially supported by the project IPN-SIP 20210291.

ones that have been benefited for drastic speed ups in their computing times [17] would be applications for: flows in porous media [5], graph compression [6], MPI combined libraries for image processing [4], query speed up in databases [16], multi-physics modelation [7], solving Boltzmann transport equations [13], CFD code speed up in non-uniform grids [19], direct modeling of gravitational fields [3], reconstructing 3D images [20], propagating acoustic waves [11], solving Lyapunov equations for control theory [8], studies of convective turbulence [2], radiative transport modelling [1], computation of Lagrangian coherent structures [9], just to mention some of them.

Due to the complexity that could represent porting applications for their GPU processing, techniques to generate parallel code through directives expressing parallelism such as OpenACC (`https://www.openacc.org/`) have been developed, which permits introducing parallelism inside the source code and delegates to the compiler the task to generate the CUDA kernel automatically. OpenACC follows the development of parallel applications paradigm like OpenMP that is oriented to multi-core architectures [18].

Despite the kindnesses offered by programming based on directives, such applications are still being developed with low-level language, basically because of the preference of taking full control in the development, as well as to avoid the dependence of compilers supporting OpenACC.

The motivation of this paper arises during the development of an application based on finite difference method to solve the heat transfer equation, to facilitate the implementation of algebraic expressions. As established by [10], the main difficulty when implementing a finite-difference code on a GPU comes from the computational stencil. For example, a fourth-order spatial operator, the thread that handles the calculation of point (i, j, k) needs to access the arrays points $(i+1, j, k)$, $(i+2, j, k)$,$(i-1, j, k)$, $(i-2, j, k)$,$(i, j+1, k)$ and so on. This implies that 13 accesses to the memory are needed on the GPU to approximate fourth-order finite difference, which is a very high value, keeping in mind that access to global memory is slow compared with shared memory. For this reason, the use of shared memory could be very convenient to minimize access latency.

The structure developed in this work consists of defining two pointers, one with multi-indices that are used to access data inside the CUDA kernel, and the other which has data in a unidimensional form to be able to copy them. This is necessary due to the GPU architecture. This paper is organized as follows. Section 2, is given the code to create bi-dimensional arrays to use them inside a kernel. Section 3, is given the code to create three-dimensional arrays to use them inside a kernel. Section 4, is applied the structures developed to the solution of the non-steady heat transfer equation by the finite difference method in 2D and 3D. Section 5, the performante test are carried out considering the use of shared memory. Finally, Section 6 presents our conclusions.

2 Bidimensional arrays

The essence to create a 2D array in CUDA so that both [][] indices can be used inside the kernel, is similar to create a continuous dynamic memory 2D array for CPU, which is shown in Listing 1.1; we use as an example the `float` type, but any primitive type could be used.

Listing 1.1: Continuous memory allocation for a 2D array in C on CPU.

```
float ** Get_Memory_Continuous_2D_float(int n, int m){
 float ** A = NULL;
 int i,j;
 cudaMallocHost((void***)&A,n*sizeof(float*));
 if (A == NULL)
  puts("Error cudaMallocHost first level"),
  exit (-1);
 cudaMallocHost((void**)&A[0],n*m*sizeof(float));
 if (A[0] == NULL) puts("Memory second level"), exit (-1);
 for (i=1; i<n; i++)
  A[i] = A[0] + i * m;
 return A;
}
```

The function shown in Listing 1.1 is a contiguous allocation of memory for a 2D array, and we shall use it as the main structure to create a 2D array in CUDA, as shown in Listing 1.2. The function `Array_2D_GPU` has 4 parameters: one pointer to a pointer variable (**), one pointer to a pointer to a pointer (***), as well as two integers. The ** (P) pointer is used to receive the address of the pointer that points to the array in its contiguous form, which shall be used to transfer data between CPU and GPU. The *** (M) pointer shall receive the address of the P pointer which shall allow the usage of two indices [][] inside the kernels, that are the other two integer kind parameters of the `Array_2D_GPU` function: `n` and `m` denote the number of rows and columns respectively.

Listing 1.2: Contiguous memory allocation in CUDA C for a 2D array on GPU.

```
void Array_2D_GPU(float **P, float ***M, int n, int m)
{
 int i; float ** P_M, ** dev_M;
 P_M = (float **) malloc(n*sizeof(float*));
 if(P_M == NULL){printf("\nMemory error"); exit(-1); }
 cudaMalloc((void**)&P_M[0],n*m*sizeof(float));
 if(P_M[0]==NULL){printf("\nMemory error"); exit(-1); }
 for (i=1; i<n; i++)
  P_M[i] = P_M[0] + i * m;
 cudaMalloc((void***)&dev_M,n*sizeof(float*));
 if(dev_M==NULL){printf("\nMemory error"); exit(-1); }
 cudaMemcpy(dev_M,P_M,n*sizeof(float*),
  cudaMemcpyHostToDevice);
 *(P) = P_M[0];
 *(M) = dev_M;
 }
```

The code shown in Listing 1.2 features two pointers: `P_M` and `dev_M`. The first one is used to transfer data between CPU and GPU, and the second one is used to handle the array using the indices ([][]) inside the kernel. The main idea behind this contiguous memory allocation for a 2D array of `m x n` is as follows:

- To create a first level pointer array (`float *`) of size `n` in CPU.
- In the first location of `P_M[0]` array, allocate memory for `m x n` elements in GPU.
- To assign to each element of `P_M[i]`, the memory address of the element in the `m x i` location (creating the 2D array).
- To create a first level pointer array (`float *`) of size `n` in GPU.
- To transfer the memory addresses assigned to `P_M[i]` to GPU, i.e. copy the content of `P_M` to `dev_M`, where `dev_M` could be used inside a CUDA kernel as a 2D array.

The last point is very important because, paradigmatically, it only considers transfers of primitive data to GPU, however, memory addresses can also be transferred.

3 Tridimensional arrays

Analogously to the 2D array creation, in the 3D case, it is also necessary to assign continuous memory. In Listing 1.3 we assign continuous memory for a 3D array in the CPU memory. Taking this code as the base, we shall build its CUDA equivalent showed in Listing 1.4.

Listing 1.3: Contiguous memory allocation in C for a 3D array on CPU.

```
float *** Get_Memory_Continuous_3D_float(int n, int m, int z)
{
 float *** A = NULL;
 float *p = NULL;
 int i,j,k;
 cudaMallocHost((void****)&A,n*sizeof(float**));
 if (A==NULL){printf("\nMemory problem"); exit(-1);}
 cudaMallocHost((void***)&A[0],n*m*sizeof(float*));
 if(A[0]==NULL){printf("\nMemory problem");exit(-1);}
 for(i=1; i<n; i++)
  A[i] = A[0] + i*m;
 cudaMallocHost((void**)&A[0][0],n*m*z*sizeof(float));
 if(A[0][0] == NULL){printf("\nMemory problem");exit(-1);}
 for(j=1; j<(n*m); j++)
  A[0][j] = A[0][0] + j*z;
 return A;
}
```

Analogous to the structure used for the 2D multi-array, we define a function `Array_3D_GPU` with five parameters (two pointers and three integers), where the pointer to a pointer `P_M` is likely used to create the structure that shall transfer data between CPU and GPU. Now `dev_M` is a `****` pointer which shall be used to handle the 3D array inside the `kernel`. The three integers, `m`, `n` and `z` are the indices of the array. The algorithm to create an array of size `m x n x z` proceed as follow:

- Allocate a second level pointers array `(float **)` of size `n` in CPU through `P_M`.
- In the first location of `P_M[0]`, to allocate memory for `m x n` pointers elements in CPU.

- To assign to each element of `P_M[i]`, the memory address of the element in the `m x i` locality.
- To create a first level pointers array (float *) of size `n` in GPU in the pointer `P_M[0][0]`.
- To assign to each element of `P_M[0][j]`, the memory address of the element in the `j x z` locality.
- In the pointer `dev_M`, to assign n elements of type `float **`.
- In the pointer `dev_M_2D`, to assign $n \times m$ elements of type `float *`. Then to transfer the $n \times m$ elements of type `float *` from `P_M[0]` to `dev_M_2D`.
- To assign `*(P) = P_M[0][0]`.
- To assign to each element `P_M[i]`, the memory address of the element in the $m \times i$ location.
- To transfer the n elements of type `float **` from `P_M` to `dev_M`.

Listing 1.4: Contiguous memory allocation in CUDA C for a 3D array on GPU.

```
void Array_3D_GPU(float ** P, float **** M, int n, int m, int z)
{
 int i, j;
 float *** P_M = NULL, *** dev_M = NULL;
 float ** dev_M_2D = NULL;
 cudaMallocHost((void****)&P_M, n*sizeof(float**));
 if(P_M == NULL){printf("\nError Host Pointers"); exit(0);}
 cudaMallocHost((void***)&P_M[0], n*m*sizeof(float*));
 if(P_M[0]==NULL){ printf("\nError Host Pointers"); exit(0); }
 for (i=1; i<n; i++)
  P_M[i] = P_M[0] + i * m;
 cudaMalloc((void**)&P_M[0][0], n*m*z*sizeof(float));
 for (j=1; j<(n*m); j++)
  P_M[0][j] = P_M[0][0] + j * z;
 cudaMalloc((void****)&dev_M, n * sizeof(float **));
 cudaMalloc((void***)&dev_M_2D, n * m * sizeof(float*));
 if(dev_M== NULL){printf("\nError dev Pointers 3D"); exit(0); }
 if(dev_M_2D==NULL){ printf("\nError dev Pointers 2D");
 exit(0); }
 cudaMemcpy(dev_M_2D, P_M[0], n*m*sizeof(float*), \ cudaMemcpyHostToDevice);
 *(P) = P_M[0][0];
 for (i=0; i<n; i++)
  P_M[i] = dev_M_2D + i * m;
 cudaMemcpy(dev_M, P_M, n*sizeof(float**), cudaMemcpyHostToDevice);
 *(M) = dev_M;
}
```

4 Application to the non-steady heat transport equation.

4.1 2D case

The bidimensional non-steady heat transport equation is expressed as,

$$\frac{\partial T}{\partial t} = C\left(\frac{\partial T^2}{\partial x^2} + \frac{\partial T^2}{\partial y^2}\right), \quad 0 \leq x, y \leq 1, t \geq 0. \tag{1}$$

Applying an explicit finite difference scheme to the Eq. (1) and rearranging terms to obtain explicitly temperature at time $l+1$, $T_{i,j}^{l+1}$ [14], we obtain the algebraic

equation:

$$T_{i,j}^{l+1} = T_{i,j}^{l} + C\Delta t \left(\frac{T_{i+1,j}^{l} - 2T_{i,j}^{l} + T_{i-1,j}^{l}}{\Delta x^2} + \frac{T_{i,j+1}^{l} - 2T_{i,j}^{l} + T_{i,j-1}^{l}}{\Delta y^2} \right). \quad (2)$$

Computational domain shall be a square that simulates a 1×1 plate, meshed with a $\Delta x = 0.003333333$ and $\Delta y = 0.005$, therefore $n_x = 301$ and $n_y = 201$, initial conditions were imposed as:

$$f(x,y) = \begin{cases} 1\text{: if } 0.05 \leq (x-05)^2 + (y-0.5)^2 \leq 0.01 \\ 0\text{: otherwise} \end{cases} \quad (3)$$

and Dirichlet boundary conditions

$$\alpha_0(y) = \alpha_1(y) = \beta_0(x) = \beta_1(x) = 0.0. \quad (4)$$

For the solution of the Eq. (2) were implemented three kernels versions named: 2D-1-1, 2D-1-2, 2D-2-2. The name structure is 2D-X-I, where X is the array dimension and I is the number of indices used inside the kernel and 2D refers to the dimension of the equation solved. T and Tn is the temperature in time l and $l+1$ respectively, and the kernels are called for every iteration in time [14].

The 2D-1-1 kernel is showed in Listing 1.5, as can be seen, determine the boundary point (perimeter points) is not trivial, in fact, if it is established n_x and n_y as the number of points in the x and y direction respectively, the elements at the boundaries can be determined with the following conditionals: $l \bmod n_x = 0$, left side; $(l-1) \bmod n_x = 0$, right side; $0 \leq l < n_x$, up side; $n_x n_y - (n_x + 1) \leq l < n_x n_y - 1$, down side; where l is the lineal index in the rows direction.

Listing 1.5: Kernel 2D using unidimensional array T with just one index l.

```
__global__ void kernel(float *T, float *Tn, int nx, int ny,\
 float dt, float C, float dx2, float dy2)
{
 int l;
 l = blockIdx.x * blockDim.x + threadIdx.x;
 float Txx;
 float Tyy;
 if(l>nx && l<((nx*ny-1)-nx) && !((l%nx)==0 || ((l+1)%nx == 0)))
 {
  Txx = (T[l+1] -2.0f*T[l] + T[l-1])/dx2;
  Tyy = (T[l+nx] -2.0f*T[l] + T[l-nx])/dy2;
  Tn[l] = T[l]+dt*C*(Txx+Tyy);
 }
}
```

In Listing 1.6, is shown the kernel 2D-1-2 that uses two indices, simplifying the conditional (`if`), to make all the calculations over the interior points, nevertheless, the boundary conditions handle is still a little bit difficult, and can be established as: $(i \times n_x + j) \bmod n_x == 0$, left side; $(i \times n_x + j - 1) \bmod n_x == 0$, right side; $0 \leq (i \times n_x + j) < n_x$, up side; $n_x n_y - (n_x + 1) \leq (i \times n_x + j) < n_x n_y - 1$, down side; where i and j are the indices.

Listing 1.6: Kernel 2D using unidimensional array T with two indices `(i,j)` .

```
__global__ void kernel(float *T, float *Tn, int ny, int \
 nx, float dt, float C,  float dx2, float dy2)
{
 int i,j,l;
 j = blockIdx.x*blockDim.x+threadIdx.x;
 i = blockIdx.y*blockDim.y+threadIdx.y;
 float Txx;
 float Tyy;
 if( (j > 0) && (j<(nx-1)) && (i > 0) &&  (i < (ny-1)))
 {
  l = i*nx+j;
  Txx = (T[l+1] -2.0f*T[l] + T[l-1])/dx2;
  Tyy = (T[l+nx] -2.0f*T[l] + T[l-nx])/dy2;
  Tn[l] = T[l]+dt*C*(Txx+Tyy);
 }
}
```

In Listing 1.7, is shown the proposed kernel 2D-2-2 using the temperature is a bidimensional array inside the kernel, to make it possible it is necessary to allocate T and Tn using the function listed in 1.2. The programability is improved and less prone to errors; the indices inside the kernel are very clear, and the boundary conditions are easily managed as: $j == 0$, left side; $j == n_x - 1$; $i == 0$, up side; $i == n_y - 1$, down side.

Listing 1.7: Kernel 2D using bidimensional array T with two indices `(i,j)`.

```
__global__ void kernel(float **T, float **Tn, int ny, int\
 nx, float dt, float C, float dx2, float dy2)
{
 int i,j;
 j = blockIdx.x*blockDim.x+threadIdx.x;
 i = blockIdx.y*blockDim.y+threadIdx.y;
 float Txx;
 float Tyy;
 if( (i > 0) && (i<(ny-1)) && (j > 0) && (j < (nx-1)))
 {
  Txx = (T[i][j+1] -2.0*T[i][j] +T[i][j-1])/dx2;
  Tyy = (T[i+1][j] -2.0*T[i][j] +T[i-1][j])/dy2;
  Tn[i][j] = T[i][j]+dt*C*(Txx+Tyy);
 }
}
```

4.2 3D case

The tridimensional non-steady heat transport equation, is expresed as,

$$\frac{\partial T}{\partial t} = C\left(\frac{\partial T^2}{\partial x^2} + \frac{\partial T^2}{\partial y^2} + \frac{\partial T^2}{\partial z^2}\right) \quad 0 \leq x, y, z \leq 1, t \geq 0 \tag{5}$$

Applying an explicit finite difference scheme to the Eq. (1) and rearranging terms to obtain explicitly temperature at time $l+1$, $T_{i,j,k}^{l+1}$:

$$T_{i,j,k}^{l+1} = T_{i,j,k}^{l} + C\Delta t \left(\frac{T_{i+1,j,k}^{l} - 2T_{i,j,k}^{l} + T_{i-1,j,k}^{l}}{\Delta x^2} + \frac{T_{i,j+1,k}^{l} - 2T_{i,j,k}^{l} + T_{i,j-1,k}^{l}}{\Delta y^2} + \frac{T_{i,j,k+1}^{l} - 2T_{i,j,k}^{l} + T_{i,j,k-1}^{l}}{\Delta z^2} \right). \tag{6}$$

In the same manner for the solution of the Eq. (6) were implemented three kernel versions named: 3D-1-1, 3D-1-3, 3D-3-3. The name structure is 3D-X-I, where X is the array dimension and I is the number of indices used inside the kernel and 3D refers to the dimension of the equation solved.

The 3D-1-1 kernel is showed in Listing 1.8, in this case, it is even more complex to determine the points on the boundary, for a 3D problem, there are 6 faces, if it is established n_x, n_y and n_z as the number of discrete points in the x, y and z direction respectively, the elements at the boundaries can be determined with the following conditionals: $l \mod n_x == 0$, left face; $(l+1) \mod n_x == 0$, right face; $0 \leq l < n_x n_y$, front face; $n_x n_y (n_z - 1) \leq l < n_x n_y n_z - 1$, back face; $l \mod n_x n_y < n_x$, top face; $l - (n_x * n_y) + nx \mod n_x n_y < n_x$, bottom face, where l is the lineal index in the rows direction.

Listing 1.8: Kernel 3D using a unidimensional array T and one index `l`. .

```
__global__ void kernel(float *T, float *Tn, int ny, int \
 nx, int nz, float dt, float C, float dx2, float dy2, float dz2)
{
 int l;
 float Txx;
 float Tyy;
 float Tzz;
 l = blockIdx.x * blockDim.x + threadIdx.x;
 if( l > (nx*ny) && l < ((nx*ny)*(nz-1)) && !((l%nx)==0 \
 || ((l+1)\%nx == 0)) && !((l%(nx*ny) < nx) || \
    (((l-(nx*ny)+nx)%((nx*ny))<nx) ) ))
 {
  Txx = (T[l+1] -2.0f*T[l] +T[l-1])/dx2;
  Tyy = (T[l+nx] -2.0f*T[l] +T[l-nx])/dy2;
  Tzz = (T[l+nx*ny] -2.0f*T[l] +T[l-nx*ny])/dz2;
  Tn[l] = T[l]+dt*C*(Txx+Tyy+Tzz);
 }
}
```

In Listing 1.9, is shown the kernel 3D-1-3 that uses three indices, improving the readability of the code, nevertheless, however boundary points have to be found as: $j + n_x(i + kn_y) \mod n_x == 0$, left face; $(j + n_x(i + kn_y) + 1) \mod n_x == 0$, right face; $0 \leq j + n_x(i + kn_y) < n_x n_y$, front face; $n_x n_y (n_z - 1) \leq j + n_x(i + kn_y) < n_x n_y n_z - 1$, back face; $j + n_x(i + kn_y) \mod n_x n_y < n_x$, top face; $j + n_x(i + kn_y) \mod n_x n_y - n_x$, bottom face, where j, i, k, are the indices in the x, y and z directions respectively.

Listing 1.9: Kernel 3D using a unidimensional array T and three indices `(i,j,k)`.

```
__global__ void kernel(float *T, float *Tn, int ny, int \
 nx, int nz, float dt, float C, float dx2, float dy2, float dz2)
```

```
{
 int i,j,k,l;
 j = blockIdx.x * blockDim.x + threadIdx.x;
 i = blockIdx.y * blockDim.y + threadIdx.y;
 k = blockIdx.z * blockDim.z + threadIdx.z;
 float Txx;
 float Tyy;
 float Tzz;
 if( (j>0) && (j<(nx-1)) && (i > 0) \
  && (i < (ny-1)) && (k>0) && (k<(nz-1)) )
 {
  l = j+i*nx+k*(nx*ny);
  Txx = (T[l+1] -2.0f*T[l] +T[l-1])/dx2;
  Tyy = (T[l+nx] -2.0f*T[l] +T[l-nx])/dy2;
  Tzz = (T[l+nx*ny] -2.0f*T[l] +T[l-nx*ny])/dz2;
  Tn[l] = T[l]+dt*C*(Txx+Tyy+Tzz);
 }
}
```

In Listing 1.10, is depicted the proposed kernel 3D-3-3, using a tridimensional array inside the kernel, to use T and Tn as tree- dimensional arrays is necessary to create the arrays using the method Listed in 1.4 , with this, the finite difference finite method is easy to implement, and the boundary conditions are easily managed as: $j == 0$, left face; $j == n_x - 1$, right face; $z == 0$, front face; $z == n_z - 1$, back face; $i == 0$, top face; $i == n_y - 1$, bottom face, where j, i, k, are the indices in the x, y and z directions respectively.

Listing 1.10: Kernel 3D using three-dimensional array T with three indices `(i,j,k)`.

```
__global__ void kernel(real ***T, real ***Tn, int ny, int nx, int nz,
real dt, real C, real dx2, real dy2, real dz2) {
  int i, j, k;
  j = blockIdx.x * blockDim.x + threadIdx.x;
  i = blockIdx.y * blockDim.y + threadIdx.y;
  k = blockIdx.z * blockDim.z + threadIdx.z;
  real Txx;
  real Tyy;
  real Tzz;
  if ((i>0) && (i<(ny - 1)) && (j>0) && (j<(nx-1)) &&
  (k>0) && (k<(nz - 1))) {
    Tzz = (T[i][j][k + 1] - 2.0 * T[i][j][k] + T[i][j][k - 1]) / dz2;
    Txx = (T[i][j + 1][k] - 2.0 * T[i][j][k] + T[i][j - 1][k]) / dx2;
    Tyy = (T[i + 1][j][k] - 2.0 * T[i][j][k] + T[i - 1][j][k]) / dy2;

    Tn[i][j][k] = T[i][j][k] + dt * C * (Txx + Tyy + Tzz);
  }
}
```

5 Performance test

It is well known that the use of shared memory to cache data in multiprocessors could improve the performance of several algorithms written in CUDA [15]. Shared memory contained in every stream multiprocessor is much faster than global memory. In fact, shared memory latency is about 100x lower than the global memory latency. Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory; in a finite difference

problem the use of share memory can increase the performance because the number of accesses to global memory are reduced.

For the kernel 2D-2-2, was added the use of shared memory. In fact, the implementation of shared memory is clear and easy to implement also providing a readable code structure. The 2D-2-2 kernel implementation with shared memory is shown in Listing 1.11 and can be seen two shared memory arrays are used, `tile` and `tilen`, the first one is used to store the data from the global memory and make the operations internally, while `tilen` have an additional shadow region to store the values required to complete the finite difference operations, thus reducing the number of accesses to global memory. For the kernel 3D-3-3, the shared memory implementation is carried out in the same way and the coding is show in Listing 1.12.

Listing 1.11: Kernel 2D using bidimensional array T with two indices `(i,j)`, and using shared memory.

```
__global__ void kernel_shared(real **T,real **Tn,int ny,int nx,real dt,
real C,real dx2,real dy2) {
  int i, j, l, m; real Txx; real Tyy;
  __shared__ real tile[tiley + 2][tilex + 2];
  __shared__ real tilen[tiley][tilex];
  j = blockIdx.x * blockDim.x + threadIdx.x;
  i = blockIdx.y * blockDim.y + threadIdx.y;
  if ((i >= 0) && (i <= (ny - 1)) && (j >= 0) && (j <= (nx - 1)))
    tile[threadIdx.y + 1][threadIdx.x + 1] = T[i][j];
  if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1))) {
    if (threadIdx.x == (blockDim.x - 1))
      tile[threadIdx.y + 1][threadIdx.x + 2] = T[i][j + 1]; // right
    if (threadIdx.y == (blockDim.y - 1))
      tile[threadIdx.y + 2][threadIdx.x + 1] = T[i + 1][j]; // down
    if (threadIdx.x == (0))
      tile[threadIdx.y + 1][threadIdx.x] = T[i][j - 1]; // left
    if (threadIdx.y == (0))
      tile[threadIdx.y][threadIdx.x + 1] = T[i - 1][j]; // up
  }
  __syncthreads();
  if ((i > 0) && (i < (ny - 1)) && (j > 0) && (j < (nx - 1))) {
    m = threadIdx.x + 1;
    l = threadIdx.y + 1;
    Txx = (tile[l][m + 1] - 2.0 * tile[l][m] + tile[l][m - 1]) / dx2;
    Tyy = (tile[l + 1][m] - 2.0 * tile[l][m] + tile[l - 1][m]) / dy2;
    tilen[l - 1][m - 1] = tile[l][m] + dt * C * (Txx + Tyy);
  }
  if (j == 0)
     tilen[threadIdx.y][threadIdx.x] = 0.0; // Dirichlet condition
  if (j == (nx - 1))
    tilen[threadIdx.y][threadIdx.x] = 0.0;
  if (i == 0)
    tilen[threadIdx.y][threadIdx.x] = 0.0;
  if (i == (ny - 1))
    tilen[threadIdx.y][threadIdx.x] = 0.0;

  if ((i >= 0) && (i <= (ny - 1)) && (j >= 0) && (j <= (nx - 1)))
    Tn[i][j] = tilen[threadIdx.y][threadIdx.x];
}
```

Listing 1.12: Kernel 3D using three-dimensional array T with three indices `(i,j,z)`, and using shared memory.

```
__global__ void kernel_shared(real ***T,real ***Tn,int ny,int nx, int nz,
```

```
real dt,real C,real dx2,real dy2,real dz2) {
int i, j, k, q, r, s;  real Txx;  real Tyy;  real Tzz;
  __shared__ real tile[tiley + 2][tilex + 2][tilez + 2];
  __shared__ real tilen[tiley][tilex][tilez];
  j = blockIdx.x * blockDim.x + threadIdx.x;
  i = blockIdx.y * blockDim.y + threadIdx.y;
  k = blockIdx.z * blockDim.z + threadIdx.z;
if ((i>=0)&&(i<=(ny-1))&&(j>=0)&&(j<=(nx - 1))&&(k>=0)&&(k<=(nz-1)))
tile[threadIdx.y+1][threadIdx.x+1][threadIdx.z+1] = T[i][j][k];
if((i>0)&&(i<(ny-1))&&(j>0)&&(j<(nx-1))&&(k>0)&&(k<(nz-1))) {
if(threadIdx.x==(0)) tile[threadIdx.y+1][threadIdx.x][threadIdx.z+1]=
T[i][j-1][k];
if(threadIdx.x==(blockDim.x-1))
tile[threadIdx.y+1][threadIdx.x+2][threadIdx.z+1]=T[i][j+1][k];
if(threadIdx.y==(0))
tile[threadIdx.y][threadIdx.x+1][threadIdx.z+1]=T[i-1][j][k];
if(threadIdx.y==(blockDim.y-1))
tile[threadIdx.y+2][threadIdx.x+1][threadIdx.z+1]=T[i+1][j][k];
if(threadIdx.z==(0))
tile[threadIdx.y+1][threadIdx.x+1][threadIdx.z]=T[i][j][k-1];
if(threadIdx.z==(blockDim.z-1))
tile[threadIdx.y+1][threadIdx.x+1][threadIdx.z+2]=T[i][j][k+1];
}
__syncthreads();
  if ((i>0)&&(i<(ny-1))&&(j>0)&&(j<(nx-1))&&(k>0)&&(k<(nz-1))) {
  q = threadIdx.x + 1; r = threadIdx.y + 1;  s = threadIdx.z + 1;
  Txx = (tile[r][q+1][s] - 2.0 * tile[r][q][s] + tile[r][q-1][s])/dx2;
  Tyy = (tile[r+1][q][s] - 2.0 * tile[r][q][s] + tile[r-1][q][s])/dy2;
  Tzz = (tile[r][q][s+1] - 2.0 * tile[r][q][s] + tile[r][q][s-1])/dz2;
  tilen[r-1][q-1][s-1] = tile[r][q][s] + dt * C * (Txx + Tyy + Tzz);
  }
  if (j == 0)
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;
  if (j == (nx - 1))
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;
  if (i == 0)
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;
  if (i == (ny - 1))
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;
  if (k == 0)
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;
  if (k == (nz - 1))
    tilen[threadIdx.y][threadIdx.x][threadIdx.z] = 0.0;
  if ((i>=0)&&(i<=(ny-1))&&(j>=0)&&(j<=(nx-1))&&(k>=0)&&(k<=(nz-1)))
    Tn[i][j][k] = tilen[threadIdx.y][threadIdx.x][threadIdx.z];
}
```

The numerical experiments were carried on a mid-range card, RTX 2060 Super with 8GB of global memory with CUDA 11.4. For the 2D Case, the Block size was configured in different sizes: 4×4, 8×4, 8×8, 16×8, 16×16, 32×16 and 32×32. The notation used is XD−I, where X is the dimension of the array used inside the kernel, and I the number of indices used and the S indicates the use of shared memory.

The computing time where obtained after 20,000 time steps, with a mesh configuration of, $301 \times 201 = 60501$, the experiments were carried out in double and single precision. The computing times were estimated in milliseconds (ms) as the average of several executions.

Figure 1 is depicted the behavior of the computing times obtained using different versions of the 2D kernels in single precision varying the threads block size (warp size), as can be observed the structure 2D-2 does not introduce latency, therefore the performance is very similar compared with the other ones versions.

The use of shared memory does not improve performance, this is because many memory moves are required in the finite difference method in proportion to the number of operations.

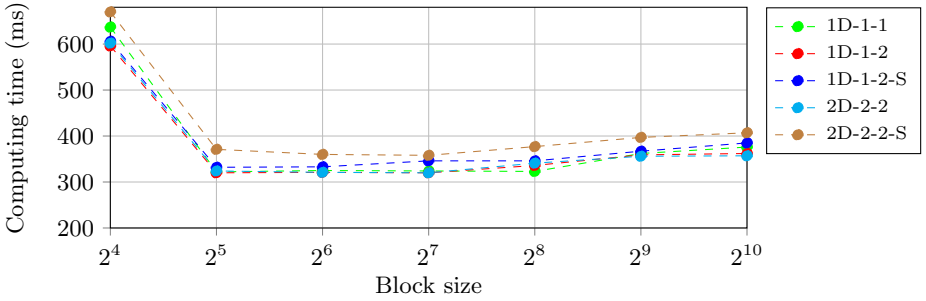


Fig. 1: Performance of the computation time in milliseconds, obtained by testing different configurations of 2D kernels in single precision.

For the double precision case, the behavior of the computing times are depicted in Figure 2, in this case there are no significant differences among the kernels, even the use of shared memory does not improve the performance.

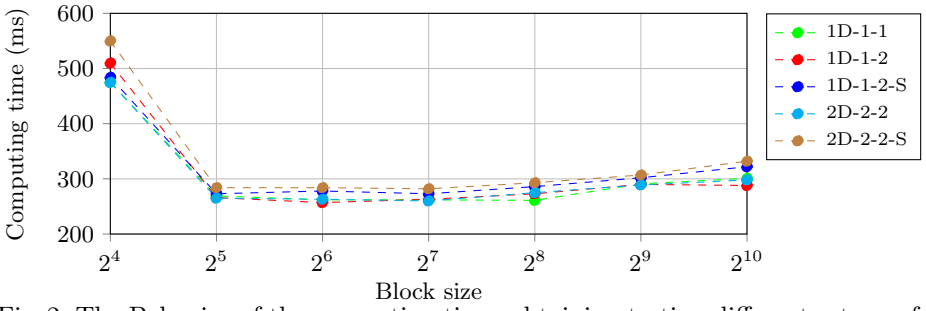


Fig. 2: The Behavior of the computing time, obtaining testing different setups of the 2D kernels in double precision.

For 3D case the sizes of the Blocks are: $4 \times 4 \times 4$, $8 \times 4 \times 4$, $8 \times 8 \times 4$, $8 \times 8 \times 8$ and $16 \times 8 \times 8$. Figure 3 depicts the behavior of the computing times for the versions of the 3D kernel and can be observed a slightly improved performance when shared memory is applied in some cases.

The computing times obtained for the double precision case are depicted in Figure 4, can be observed, a slight drop in performance for the structure proposed when the block size is 2^9, however, the performance is very similar in all the cases, even using shared memory.

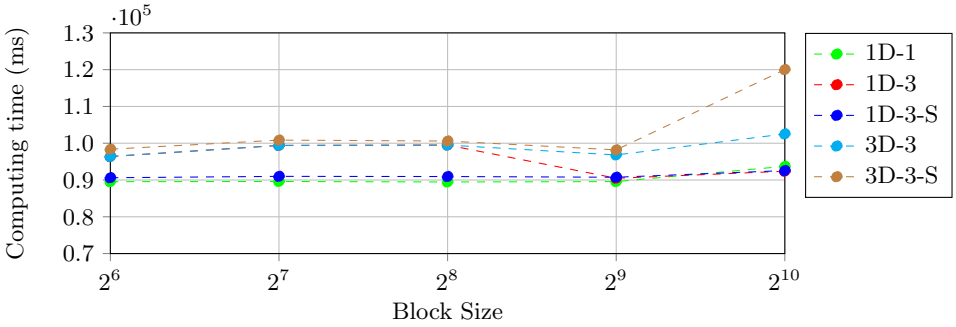


Fig. 3: Computational time performance in milliseconds, obtained by testing different configurations of the 3D kernels in single precision.

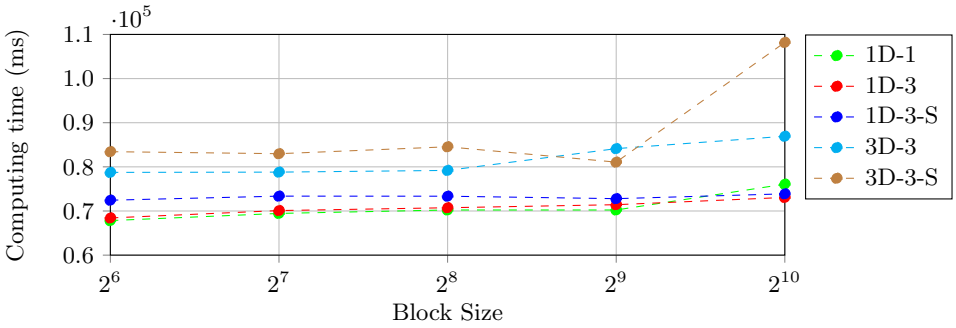


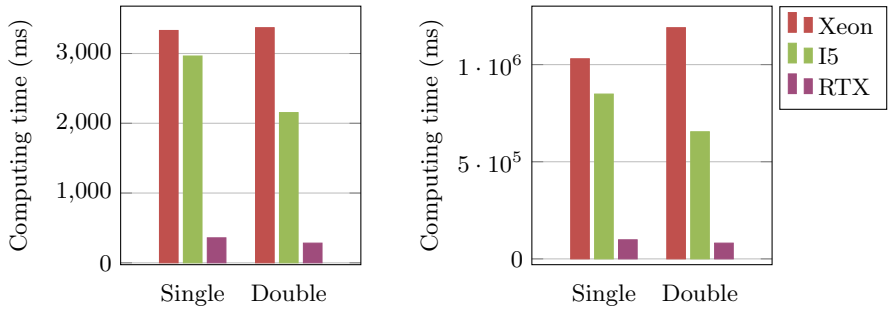
Fig. 4: Performance of the computation time in milliseconds, obtained by testing different configurations of the 3D kernels in double precision.

Finally, we compared the computational times obtained with the kernels: 2D-2-2-S with a block size of (16×8), and 3D-3-3-S with a block size of $(8 \times 8 \times 8)$, against its sequential counterpart with two Intel processors, a Xeon E5-2630V4 and an I5-4200U. For the 2D case, the results show that the CUDA code running on the RTX 2060 card is 90.30X and 11.96X faster in single and double precision respectively with respect to the serial code running on the Xeon processor, and 8.27X and 7.63X faster than the I5 in single and double precision respectively.

For the 3D case, the CUDA code is 10.49X and 14.67X faster in single and double precision respectively with respect to the Xeon processor. With respect to the I5 8.64X and 8.07X times faster in single and double precision respectively (see Figure 5).

6 Conclusions

This paper introduced a data structure that allows creating multi-arrays in CUDA as it is done in standard C language. It consists of defining an auxil-



(a) 2D Heat Transfer (b) 3D Heat Transfer

Fig. 5: Comparison between computation times and both single and double precision (Intel Xeon E52630V, Intel 5I5-4200U, and RTX2060 card), for 2D case (a) and 3D case (b).

iary pointer to transfer data between CPU and GPU. This structure improves the readability of the source code because it is able to handle `[][]` or `[][][]` indices for 2D and 3D arrays respectively, easing the maintenance and modification of the application, especially in the management of the boundary conditions. The solution of the heat transport equation

Also, the shared memory usage is introduced to probe its behavior and computational benefit multi-dimensional array inside a kernel, showing certain advantages concerning performance in some cases.

With the codelets introduced in this work, we offer an options to question found in the fora like `https://forums.developer.nvidia.com/t/passing-a-multidimensional-array-to-kernel-how-to-allocate-space-in-host-and-pass-to-device/10853`

Finally, it is necessary to mention that the performance results obtained, could vary, depending on the architecture and application.

References

1. Al-Refaie, A.F., Yurchenko, S.N., Tennyson, J.: **G**PU **A**ccelerated **IN**tensities MPI (GAIN-MPI): A new method of computing Einstein-A coefficients. Computer Physics Communications **214**, 216 – 224 (2017)
2. Calore, E., Gabbana, A., Kraus, J., Pellegrini, E., Schifano, S., Tripiccione, R.: Massively parallel lattice–Boltzmann codes on large GPU clusters. Parallel Computing **58**, 1 – 24 (2016)
3. Couder-Castañeda, C., Ortiz-Alemán, C., Orozco-Del-Castillo, M., Nava-Flores, M.: Tesla gpus versus mpi with openmp for the forward modeling of gravity and gravity gradient of large prisms ensemble. Journal of Applied Mathematics **2013** (2013)
4. Galizia, A., D'Agostino, D., Clematis, A.: An mpi-cuda library for image processing on hpc architectures. Journal of Computational and Applied Mathematics **273**, 414–427 (2015)

5. Huang, C., Shi, B., He, N., Chai, Z.: Implementation of multi-gpu based lattice boltzmann method for flow through porous media. Advances in Applied Mathematics and Mechanics **7**(1), 1–12 (2015)
6. Kaczmarski, K., Przymus, P., Rzazewski, P.: Improving high-performance gpu graph traversal with compression. Advances in Intelligent Systems and Computing **312**, 201–214 (2015)
7. Krol, D., Harris, J., Zydek, D.: Hybrid gpu/cpu approach to multiphysics simulation. Advances in Intelligent Systems and Computing **1089**, 893–899 (2015)
8. Köhler, M., Saak, J.: On GPU acceleration of common solvers for (quasi-) triangular generalized Lyapunov equations. Parallel Computing **57**, 212 – 221 (2016)
9. Lin, M., Xu, M., Fu, X.: GPU-accelerated computing for Lagrangian coherent structures of multi-body gravitational regimes. Astrophysics and Space Science **362**(4), 66 (2017). https://doi.org/10.1007/s10509-017-3050-y
10. Michéa, D., Komatitsch, D.: Accelerating a three-dimensional finite-difference wave propagation code using gpu graphics cards. Geophysical Journal International **182**(1), 389–402 (2010)
11. Nakata, N., Tsuji, T., Matsuoka, T.: Acceleration of computation speed for elastic wave simulation using a graphic processing unit. Exploration Geophysics **42**(1), 98–104 (2011)
12. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., Purcell, T.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum **26**(1), 80–113 (2007)
13. Priimak, D.: 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 (2014)
14. Recktenwald, G.W.: Finite-difference approximations to the heat equation. Mechanical Engineering **10**(01) (2004)
15. Rosen, P.: A visual approach to investigating shared and global memory behavior of cuda kernels. Computer Graphics Forum **32**(3 PART2), 161–170 (2013). https://doi.org/10.1111/cgf.12103
16. Strohm, P., Wittmer, S., Haberstroh, A., Lauer, T.: Gpu-accelerated quantification filters for analytical queries in multidimensional databases. Advances in Intelligent Systems and Computing **312**, 229–242 (2015)
17. Vuduc, R., Czechowski, K.: What gpu computing means for high-end systems. system **8**, 10 (2011)
18. Wienke, S., Terboven, C., Beyer, J.C., Müller, M.S.: A pattern-based comparison of openacc and openmp for accelerator computing. In: Euro-Par 2014 Parallel Processing, pp. 812–823. Springer (2014)
19. Xu, C., Deng, X., Zhang, L., Fang, J., Wang, G., Jiang, Y., Cao, W., Che, Y., Wang, Y., Wang, Z., Liu, W., Cheng, X.: 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 (2014)
20. Zhang, T., Du, Y., Huang, T., Li, X.: Gpu-accelerated 3d reconstruction of porous media using multiple-point statistics. Computational Geosciences (2014)

GLOSARIO

API un conjunto de funciones y procedimientos que permiten la creación de aplicaciones que acceden a las características o datos de un sistema operativo, aplicación u otro servicio.. 26, 37, 38

ARM ARM es una arquitectura RISC (Reduced Instruction Set Computer, Ordenador con Conjunto Reducido de Instrucciones) de 32 o 64 bits desarrollada por ARM Holdings. Se llamó Advanced RISC Machine, y anteriormente Acorn RISC Machine. La arquitectura ARM es el conjunto de instrucciones de 32 bits más ampliamente utilizado en unidades producidas. Concebida originalmente por Acorn Computers para su uso en ordenadores personales, los primeros productos basados en ARM eran los Acorn Archimedes, lanzados en 1987.. 9

búfer Búfer es una memoria de corto plazo de una persona. Es aquel almacenamiento que guarda pequeños datos o movimientos que se realizan dentro de una computadora, básicamente para optimizar el tiempo de respuesta del procesador. Este tipo de memorias no procesan completamente la data disponible para la función, se basan en las respuestas rápidas y la carga de archivos necesarios para la visualización de un documento o escuchar un archivo de música.. 21

chip El término chip hace referencia a un elemento muy pequeño, fabricado con un material semiconductor, que presenta numerosos circuitos integrados. Estos circuitos le permiten desarrollar diversas funciones en aparatos electrónicos.. 8, 25

cluster Conjunto de equipos de cómputo que se comportan como una Supercomputadora única. Son utilizados principalmente para la solución de problemas de alto costo computacional referentes a las ciencias, las ingenierías y el comercio.. 6–10, 20, 22, 23

Device Dispositivo CUDA significa dispositivos habilitados para CUDA. También puede ser una GPU habilitada para CUDA o algún otro dispositivo.. 27

GigaFLOP GigaFLOP es una unidad de medida que se usa para medir el rendimiento de la unidad de punto flotante de una computadora, comúnmente conocida como FPU. Un GigaFlop equivale a mil millones (1.000.000.000) de FLOPS, u operaciones de punto flotante, por segundo.. 22

hilo En informática, un hilo de ejecución es la secuencia de instrucciones programadas que un programador puede administrar de forma independiente, que

generalmente es parte del sistema operativo. La implementación de subprocesos y procesos difiere entre sistemas operativos, pero en la mayoría de los casos un subproceso es un componente de un proceso. Puede haber varios subprocesos dentro de un proceso, ejecutándose simultáneamente y compartiendo recursos como la memoria, mientras que diferentes procesos no comparten estos recursos. En particular, los hilos de un proceso comparten su código ejecutable y los valores de sus variables asignadas dinámicamente y las variables globales.. 29, 30

Host En CUDA, el host se refiere a la CPU y su memoria, mientras que el dispositivo se refiere a el GPU y su memoria. El código que se ejecuta en el host puede administrar la memoria tanto en el host como en el dispositivo, y también lanza kernels que son funciones que se ejecutan en el dispositivo.. 27, 32, 33, 35, 38–42

HPC La computación de alto rendimiento (HPC) es la capacidad de procesar datos y realizar cálculos complejos a altas velocidades. Para ponerlo en perspectiva, una computadora portátil o de escritorio con un procesador de 3 GHz puede realizar alrededor de 3 mil millones de cálculos por segundo. Si bien eso es mucho más rápido de lo que cualquier ser humano puede lograr, palidece en comparación con las soluciones de HPC que pueden realizar billones de cálculos por segundo. Uno de los tipos más conocidos de soluciones HPC es la supercomputadora. Una supercomputadora contiene miles de nodos de cálculo que trabajan juntos para completar una o más tareas. A esto se le llama procesamiento paralelo. Es similar a tener miles de PC conectados en red, combinando la potencia informática para completar las tareas más rápido.. 9, 21, 22

HyperThreading Hyperthreading (HT), es el nombre de Intel para el multiproceso simultáneo. Básicamente significa que un núcleo de CPU puede trabajar en dos problemas al mismo tiempo (Hilos). No significa que la CPU pueda hacer el doble de trabajo. Solo que puede garantizar que se utilice toda su capacidad al lidiar con múltiples problemas más simples a la vez.. 30

MIMD (Multiple Instructions Multiple Data) múltiples instrucciones, múltiples datos.. 7

MISD (Multiple Instructions Single Data) múltiples instrucciones, un solo dato (arquitectura no existente).. 7

PCI-Express PCIe (interconexión rápida de componentes periféricos) es un estándar de interfaz para conectar componentes de alta velocidad. Cada placa base de PC de escritorio tiene varias ranuras PCIe que puede usar para agregar GPU (también conocidas como tarjetas de video, también conocidas como tarjetas gráficas), tarjetas RAID, tarjetas Wi-Fi o tarjetas adicionales SSD (unidad de estado sólido). Las ranuras PCIe vienen en diferentes configuraciones físicas: x1, x4, x8, x16, x32. El número después de la x le indica cuántos

carriles (cómo viajan los datos hacia y desde la tarjeta PCIe) tiene la ranura PCIe. Una ranura PCIe x1 tiene un carril y puede mover datos a un bit por ciclo. Una ranura PCIe x2 tiene dos carriles y puede mover datos a dos bits por ciclo (y así sucesivamente). 31

POSIX En primer lugar, POSIX significa Portable Operating System Interface. Consiste en una familia de estándares especificadas por la IEEE con el objetivo de facilitar la interoperabilidad de sistemas operativos. Además, POSIX establece las reglas para la portabilidad de programas. Por ejemplo, cuando se desarrolla software que cumple con los estándares POSIX existe una gran probabilidad de que se podrá utilizar en sistemas operativos del tipo Unix. Si se ignoran tales reglas, es muy posible que el programa o librería funcione bien en un sistema dado pero que no lo haga en otro.. 16

programabilidad La programabilidad generalmente se refiere a la lógica del programa (reglas de negocio), pero también se refiere al diseño de la interfaz de usuario que incluye las opciones de menús, botones y cuadros de diálogo.. 9

PTX PTX proporciona un modelo de programación estable y un conjunto de instrucciones para la programación paralela de propósito general. Está diseñado para ser eficiente en las GPU NVIDIA que admiten las funciones de cálculo definidas por la arquitectura NVIDIA Tesla. Los compiladores de lenguaje de alto nivel para lenguajes como CUDA y C / C ++ generan instrucciones PTX, que están optimizadas y traducidas a instrucciones de arquitectura de destino nativas.. 25

runtime El runtime es el período de tiempo en el que se ejecuta un programa. Comienza cuando se abre (o ejecuta) un programa y finaliza cuando el programa se cierra o finaliza. runtime es un término técnico que se utiliza con mayor frecuencia en el desarrollo de software y se utiliza comúnmente en el contexto de un *error de tiempo de ejecución*, que es un error que ocurre mientras se ejecuta un programa. El término .erroror de tiempo de ejecución"se utiliza para distinguir de otros tipos de errores, como errores de sintaxis y errores de compilación, que se producen antes de ejecutar un programa.. 38

SIMD (Single Instruction Multiple Data) una instrucción, múltiples datos.. 7

SISD (Single Instruction Single Data) una instrucción, un solo dato.. 7

SMP Una computadora de procesamiento simétrico contiene al menos 2 procesadores que trabajan en conjunto, también se les conoce como computadoras de memoria compartida.. 6, 7, 9, 10

socket El zócalo de CPU (socket en inglés) es un tipo de zócalo electrónico (sistema electromecánico de soporte y conexión eléctrica) instalado en la placa base, que se usa para fijar y conectar el microprocesador, sin soldarlo lo cual permite ser extraído después.. 8, 9

TeraFLOP TeraFLOPS es un término que se utiliza en informática para medir la potencia de cálculo de una CPU o GPU. La palabra TeraFLOPS se forma con el prefijo Tera (que significa billón europeo o trillón americano), y el acrónimo FLOPS (Floating Point Operations Per Second) que significa: operaciones de coma flotante por segundo.. 22

BIBLIOGRAFÍA

Akl, S. G. (2000). Parallel real-time computation: Sometimes quantity means quality, *Proceedings International Symposium on Parallel Architectures, Algorithms and Networks. I-SPAN 2000*, IEEE, pp. 2–11.

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.

Baker, M. and Buyya, R. (1999). Cluster computing at a glance, *High Performance Cluster Computing: Architectures and Systems* **1**(3-47): 12.

Blazewicz, M., Brandt, S. R., Diener, P., Koppelman, D. M., Kurowski, K., Löffler, F., Schnetter, E. and Tao, J. (2012). A massive data parallel computational framework for petascale/exascale hybrid computer systems, *arXiv preprint arXiv:1201.2118* .

Buck, I. (2007). Gpu computing with nvidia cuda, *ACM SIGGRAPH 2007 courses*, pp. 6–es.

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, *Seventh IEEE international symposium on cluster computing and the grid (CCGrid'07)*, IEEE, pp. 471–478.

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

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-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**.

Dehal, R. S., Munjal, C., Ansari, A. A. and Kushwaha, A. S. (2018). Gpu computing revolution: Cuda, *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pp. 197–201.

Evrard, A. E., MacFarland, T., Couchman, H., Colberg, J., Yoshida, N., White, S., Jenkins, A., Frenk, C., Pearce, F., Peacock, J. et al. (2002). Galaxy clusters in hubble volume simulations: cosmological constraints from sky survey populations, *The Astrophysical Journal* **573**(1): 7.

Forsythe, G. E. and Wasow, W. R. (1960). Finite difference methods, *Partial Differential* .

Foster, I. (2020). *Designing and building parallel programs: concepts and tools for parallel software engineering*, Addison-Wesley.

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.

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.

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.

Kindratenko, V. V., Enos, J. J., Shi, G., Showerman, M. T., Arnold, G. W., Stone, J. E., Phillips, J. C. and Hwu, W.-m. (2009). Gpu clusters for high-performance computing, *2009 IEEE International Conference on Cluster Computing and Workshops*, IEEE, pp. 1–8.

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

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P. et al. (2010). Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu, *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 451–460.

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.

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.

Narasimhan, T. N. (1999). Fourier's heat conduction equation: History, influence, and connections, *Reviews of Geophysics* **37**(1): 151–172.

Norgan, A. P., Coffman, P. K., Kocher, J.-P. A., Katzmann, D. J. and Sosa, C. P. (2011). Multilevel parallelization of autodock 4.2, *Journal of cheminformatics* **3**(1): 12.

NVIDIA (2010). Ptx: Parallel thread execution isa version 2.3, *Dostopno na: http://developer. download. nvidia. com/compute/cuda* **3**.

Nvidia, C. (2007). Compute unified device architecture programming guide.

Owens, J. (2007). Gpu architecture overview, *ACM SIGGRAPH 2007 courses*, pp. 2–es.

Piedra, S., Torres, M. and Ledesma, S. (2019). Thermal numerical analysis of the primary composite structure of a cubesat, *Aerospace* **6**(9): 97.

Pletcher, R. H., Tannehill, J. C. and Anderson, D. (2012). *Computational fluid mechanics and heat transfer*, CRC press.

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.

Rabenseifner, R., Hager, G. and Jost, G. (2009). Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes, *2009 17th Euromicro international conference on parallel, distributed and network-based processing*, IEEE, pp. 427–436.

Rodríguez-Sánchez, A., Couder-Castañeda, C., Hernández-Gómez, J., Medina, I., Peña-Ruiz, S., Sosa-Pedroza, J. and Enciso-Aguilar, M. (2018). Analysis of electromagnetic propagation from mhz to thz with a memory-optimised cpml-fdtd algorithm, *International Journal of Antennas and Propagation* **2018**.

Rosen, P. (2013). A visual approach to investigating shared and global memory behavior of cuda kernels, *Computer Graphics Forum* **32**(3 PART2): 161–170.

Ross, P. E. (2008). Why cpu frequency stalled, *IEEE Spectrum* **45**(4): 72–72.

Snyder, L. (1988). A taxonomy of synchronous parallel machines, *Technical report*, WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE.

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.

Sunderam, V. S. (1990). Pvm: A framework for parallel distributed computing, *Concurrency: practice and experience* **2**(4): 315–339.

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, *2014 IEEE 28th international parallel and*

distributed processing symposium, IEEE, pp. 1063–1072.

van de Geijn, R. and Goto, K. (2011). *BLAS (Basic Linear Algebra Subprograms)*, Springer US, Boston, MA, pp. 157–164.

Vuduc, R. and Czechowski, K. (2011). What gpu computing means for high-end systems, *IEEE Micro* **31**(4): 74–78.

Winkler, D., Meister, M., Rezavand, M. and Rauch, W. (2017). gpusphase—a shared memory caching implementation for 2d sph using cuda, *Computer Physics Communications* **213**: 165–180.

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* .

Zhou, P.-b. (1993). *Finite Difference Method*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 63–94.

Ziabari, A. K., Sun, Y., Ma, Y., Schaa, D., Abellán, J. L., Ubal, R., Kim, J., Joshi, A. and Kaeli, D. (2016). Umh: A hardware-based unified memory hierarchy for systems with multiple discrete gpus, *ACM Transactions on Architecture and Code Optimization (TACO)* **13**(4): 1–25.