Universidad Nacional Autónoma de México

Facultad de Ciencias

ALGORITMOS DIVIDE Y VENCERÁS EN LAS GPU.

TESIS

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

ARMANDO BALLINAS NANGUËLÚ

TUTOR:
DR. JOSÉ DAVID FLORES PEÑALOZA

2015

Ciudad Universitaria, D. F.

1. Datos del alumno
    - Armando Ballinas Nangüelú
    - 56 80 63 41
    - Universidad Nacional Autónoma de México
    - Facultad de Ciencias
    - Ciencias de la Computación
    - 306001073
2. Datos del tutor
    - Dr. José David Flores Peñaloza.
3. Datos del sinodal 1
    - Dr. Armando Castañeda Rojano.
4. Datos del sinodal 2
    - Dr. José de Jesús Galaviz Casas.
5. Datos del sinodal 3
    - Dr. Sergio Rajsbaum Gorodezky.
6. Datos del sinodal 4
    - Dra. María de Luz Gasca Soto.
7. Datos del trabajo escrito
    - Algoritmos *divide y vencerás* en las GPU.
    - p 86.
    - 2015

Dedicado a las dos mujeres que me han hecho ser lo que soy: mi madre y mi abuelita.

Agradecimientos

A mi familia: mi mamá Marlene y mi abuelita Celia por haberme transformado en una persona sensata y responsable, por haberme cuidado y protegido siempre y por darme siempre lo mejor de ellas.

A Miguel Ángel Romero por ser mi mejor amigo, por esas incontables y largas pláticas de practicamente cualquier tema, por escuchar mis historias e incoherencias y por siempre estar allí para mi. También agradezco a la familia Romero por hacerme sentir parte de ella y siempre apoyarme; con ellos, me siento como en casa.

A Diana Montes por ser esa amiga a la que le puedo contar lo que sea, por ser la que escucha, pero también entiende. Gracias por esas pláticas, consejos y palabras que siempre han estado ahí cuando las he necesitado.

A Estefanía Prieto por ser una persona muy especial e importante en mi vida. Gracias por siempre estar conmigo y hacerme sentir bien. Gracias por compartir grandes momentos conmigo, por quererme y por cuidarme siempre.

A Manuel Alcántara, Renato Zamudio y Karla Vargas por esas increíbles salidas y memorables reuniones. Gracias por esas tardes de juegos de mesa; que, aunque no gane, me divierten, entretienen y me hacen sentir pleno al compartirlas con ustedes.

A Luis Ruiz ("boss") por ser un gran amigo y compañero durante la carrera. Gracias por ser una excelente persona y un gran compañero.

A mis amigos Raúl Sandoval, Sarahí Romano, Enrique Bernal y Antonio Gómez por ser muy importantes en distintas etapas de mi vida. Gracias por platicar conmigo, entenderme y hacer mis tardes más divertidas.

A mi tutor el Dr. David Flores por sus consejos y enseñanzas y por guiarme en este camino de la investigación en las Ciencias de la Computación.

A todos los profesores que me han dejado más que los conocimientos impartidos durante sus clases; en especial al Dr. Héctor Méndez , al Dr. César Hernández y al Dr. Favio Miranda porque, con su excelsa manera de enseñar, me impulsaron a ser profesor y a transmitir mis conocimientos a otras personas. Espero algún día hacerlo tan bien como ellos.

A mis sinodales el Dr. Armando Castañeda, el Dr. José Galaviz, el Dr. Sergio Rajsbaum y la Dra. María de Luz Gasca por sus invaluables comentarios y correcciones a este trabajo.

Tabla de contenidos.

Lista de figuras.

Lista de Algoritmos.

Introducción.

En este trabajo se presentan estrategias de adaptación de algoritmos con paradigma *divide y vencerás* (en inglés estos algoritmos se llaman *divide and conquer algorithms*) a arquitecturas de procesamiento gráfico conocidas por sus siglas en inglés como GPU (*Graphical Processor Unit*). También se evalúa la eficiencia de recursos de cómputo mediante pruebas de rendimiento ante implementaciones secuenciales.

La principal motivación de este trabajo surgió tras haber aprendido a programar las GPU en un seminario de la Facultad de Ciencias de la UNAM. Después de concluir con este seminario se siguió trabajando con ellas y así se pensó en estudiar algoritmos con este paradigma adaptando las versiones secuenciales a la arquitectura de las GPU. En particular en este trabajo se trabaja con la arquitectura CUDA[1] (CUDA significa, por sus siglas en inglés, *Compute Unified Device Architecture*) de la marca NVIDIA[2].

El trabajo se divide en dos secciones. En la primera, compuesta por los primeros cuatro capítulos, se presenta el marco teórico del trabajo. Mientras que en la segunda sección, compuesta por los siguientes cuatro capítulos, se presentan casos de estudio en los cuales se expone y se explica un problema estudiado en detalle.

En el Capítulo 1 se explica la historia de las GPU y porqué son importantes hoy en día, así como sus principales aplicaciones tanto en el ámbito científico como en el comercial. También se encuentra la historia de la arquitectura CUDA de NVIDIA y la motivación por usarla en este trabajo.

En el Capítulo 2 se expone el lenguaje de programación CUDA C. Así como las características principales de este lenguaje y la manera de crear programas en CUDA C.

En el Capítulo 3 se introduce el paradigma *divide y vencerás*. A lo largo del capítulo se exponen las características de este tipo de algoritmos y la gran variedad de usos que se le han dado a este paradigma. Así como la manera en que se modifica la técnica de implementación inicial para adecuarla a las GPU.

En el Capítulo 4 se presenta de manera teórica el paralelismo que tienen los distintos algoritmos *divide y vencerás*; en particular, se discute el paralelismo precalculable contra el paralelismo no precalculable y el impacto de esto en el diseño y la implementación de un algoritmo *divide y vencerás*.

En la segunda sección del trabajo se consideran cuatro casos de estudio; es decir, cuatro

[1] CUDA® es una marca registrada de NVIDIA Corporation.
[2] NVIDIA® es una marca registrada de NVIDIA Corporation.

algoritmos con paradigma *divide y vencerás*. Para cada uno de estos algoritmos se expone el algoritmo secuencial y su complejidad asintótica. A continuación, se consideran una serie de observaciones importantes para obtener la versión paralela. Después, se encuentra el algoritmo paralelo adecuado a CUDA y, finalmente, se analiza la eficiencia por medio de comparaciones de rendimiento contra la versión secuencial u otras versiones en CUDA del algoritmo.

En el Capítulo 5 se presenta el primer caso de estudio. Siendo este el algoritmo de la transformada rápida de Fourier (en inglés *Fast Fourier Transform*). Este algoritmo tiene muchas aplicaciones y es muy usado sobretodo en procesamiento digital de señales. En este trabajo se presenta otra aplicación pues se usa este algoritmo para multiplicar polinomios de manera eficiente.

En el Capítulo 6 se encuentra la subrutina de mezcla (en inglés *merge*) del algoritmo de ordenación por mezcla (en inglés *mergesort*). En este capítulo se discute una implementación paralela de dicha subrutina en lugar de la forma secuencial utilizada comúnmente.

En el Capítulo 7 se encuentra el algoritmo de ordenación por mezcla. Este es un conocido algoritmo para ordenar una secuencia de números arbitrarios de manera óptima y tiene el paradigma *divide y vencerás* además de la estrategia de mezcla vista en el capítulo anterior.

En el Capítulo 8 se tiene el último caso de estudio que consiste en trabajar con el algoritmo de la pareja de puntos más cercana (en inglés *closest pair of points*). Este algoritmo encuentra la pareja de puntos más cercana, es decir, con la menor distancia entre ellos dentro de un conjunto de puntos arbitrario en el plano.

Por último, se exponen las conclusiones generales obtenidas tras haber realizado los análisis de los casos de estudio y se da un panorama general del trabajo que se podría realizar en el futuro en este campo de estudio.

Capítulo 1

Arquitectura de las GPU.

En este capítulo se describe la evolución de los dispositivos gráficos y de las distintas arquitecturas CUDA. En la primera sección se plantea la motivación para crear dispositivos de este tipo. En la segunda se relata la historia de las tarjetas NVIDIA y en las siguientes se muestran las arquitecturas más recientes.

1.1. De procesadores rápidos a procesadores multi-hilo.

Hasta hace 10 años los desarrolladores hacían software teniendo en consideración que la velocidad de los procesadores aumentaba con cada nueva generación, sin embargo esto dejó de ser cierto en 2003 pues en ese entonces se había alcanzado una limitante física importante: si se aumentaba la frecuencia de los procesadores pero se preservaba el número de transistores en ellos entonces el calor que se generaba dentro del chip aumentaba drásticamente y esto representaba un riesgo para el procesador. Pero, es aún más importante considerar que el paralelismo a nivel de instrucción ya no se podía (ni puede) mejorar mucho, por lo que se debía estudiar el paralelismo a nivel de hilo.

Es entonces cuando los diseñadores de procesadores empezaron a optar por dos estrategias. La primera llamada *multi-núcleo* (en inglés *multi-core*) que busca mantener el tiempo de ejecución alcanzado por los programas secuenciales mientras se modifican para funcionar con varios núcleos. Los procesadores multi-núcleo comenzaron con dos núcleos y este número ha ido aumentando con cada generación nueva de semiconductores. Este es el modelo que adoptaron las principales compañías de diseño de procesadores como Intel[1]o AMD[2]. Hoy en día las computadoras personales tienen procesadores con dos, cuatro u ocho núcleos, mientras que los procesadores para servidores llegan hasta 18 núcleos. Cabe mencionar que debido a esto los términos núcleo y procesador se pueden intercambiar, pues aunque físicamente hay un solo procesador donde residen varios núcleos, cada núcleo tiene la capacidad para ejecutar un programa, tal y como lo hacen los procesadores con un sólo núcleo.

[1]Intel® es una marca registrada de Intel Corporation.
[2]AMD® es una marca registrada de Advanced Micro Devices, Inc.

El otro modelo llamado *multi-hilo* se enfoca en mejorar el desempeño general de las aplicaciones paralelas. Las compañías creadoras de tarjetas de video como NVIDIA o ATI (hoy en día AMD) utilizaron esta estrategia. Estos dispositivos comenzaron con un gran número de hilos y han ido aumentando con cada generación. Por ejemplo, la tarjeta GeForce[3] GTX 780 Ti cuenta con 2880 núcleos CUDA, alternando la ejecución, con costo administrativo despreciable, de hasta 2048 hilos por cada uno de los 65536 bloques simultáneos que se pueden lanzar.

A diferencia de los núcleos residentes en un procesador multi-núcleo, los núcleos de una tarjeta de video no son tan robustos y complejos. A veces se dice que estos núcleos son "poco más que una ALU". Es decir, aunque pueden ejecutar instrucciones no poseen todas las características que poseen los núcleos de un procesador multi-núcleo por lo que son mucho más baratos de fabricar y se pueden incrustar miles de ellos en una tarjeta de video.

Desde hace años el modelo multi-hilo ha mejorado considerablemente el rendimiento neto de aplicaciones con punto flotante. Esto se debe al distinto enfoque que tienen ambas estrategias. Por un lado, los procesadores multi-núcleo están enfocados en optimizar código secuencial. Por ejemplo, en una computadora de dos núcleos un núcleo puede ejecutar el programa que el usuario esté usando actualmente, mientras que el otro ejecuta procesos del sistema operativo o de otros programas logrando que el programa del usuario no pierda rendmiento debido a los cambios de contexto del procesador. Por otro lado, los procesadores multi-hilo se enfocan en tareas que requieren un gran número de operaciones y alto grado de paralelismo. Un ejemplo de lo anterior es un programa que le aplica un filtro de escala de grises a una imagen a color pues la manipulación de cada pixel es independiente por lo que de manera paralela e independiente se puede calcular el tono de gris de cada pixel.

1.2. De las tarjetas de video a las GPU.

En esta sección se describe la historia de las tarjetas de video y cómo fueron evolucionando hasta convertirse en una GPU de hoy en día.

1.2.1. Evolución de los *pipelines* gráficos.

Entre los años ochenta y noventa las herramientas para graficar en tercera dimensión pasaron de ser sistemas grandes y costosos a ser aceleradores pequeños para computadoras personales. Durante este periodo no solo los precios disminuyeron drásticamente sino también aumentó considerablemente el poder de cómputo. Se pasó de procesar 50 millones de pixeles por segundo a procesar casi 1000 millones de pixeles por segundo.

Durante los años noventa el hardware de gráficos que lideraba el mercado estaba compuesto por *pipelines* que tenían funciones predefinidas que eran configurables pero no programables. Los *pipelines* se componen por varias etapas a nivel de hardware, donde cada etapa se encarga de una única tarea en específico.

[3]GeForce® es una marca registrada de NVIDIA Corporation.

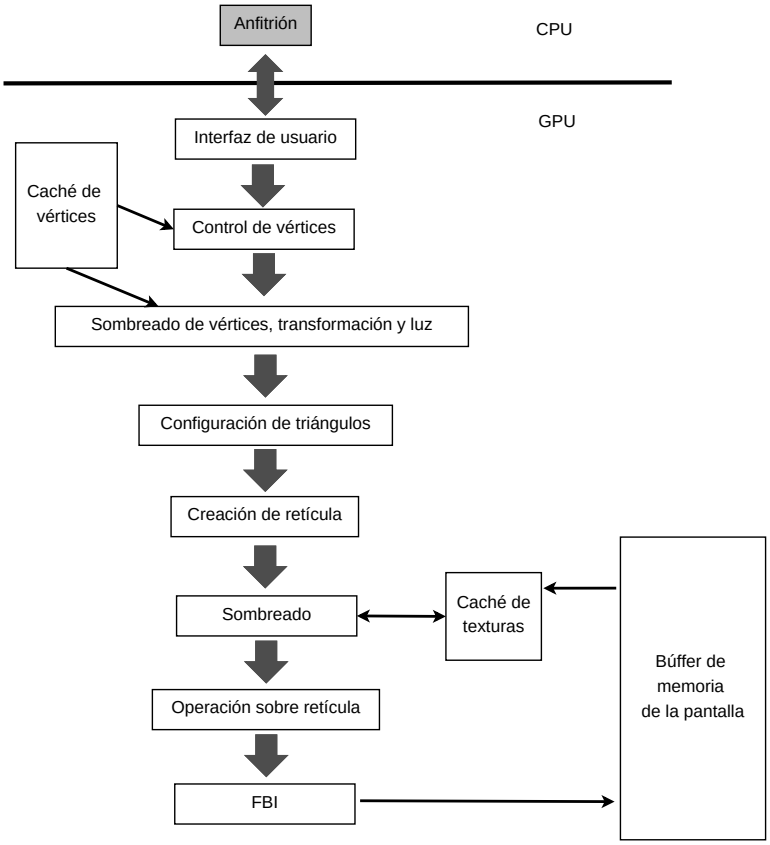

Figura 1.1: Imagen de un *pipeline* gráfico de NVIDIA.

En este tiempo se popularizaron las *API* (siglas en inglés para *Application Programming Interface*) de bibliotecas que permitían configurar ciertas opciones en el *pipeline* de manera clara y sencilla . Las API para gráficos populares, usadas hasta hoy en día son *DirectX*[4] de Microsoft[5] y *OpenGL* que es código abierto. Estas API permiten, entre otras cosas, enviar comandos y datos a las tarjetas de video para que se muestren en pantalla.

En la Figura 1.1 se muestran las diferentes etapas que componen a un *pipeline* gráfico de NVIDIA y a continuación se decribe brevemente lo que hace cada etapa.

La primera etapa es la interfaz de usuario que es la que recibe los comandos del CPU y se encarga de transferir los datos sin procesar del CPU a la tarjeta, esto usualmente utilizando DMA (DMA son las siglas en inglés para *Direct Memory Access*) que es una técnica para copiar datos entre un dispositivo de entrada y salida y la memoria DRAM.

Como los *pipelines* gráficos están diseñados para dibujar triángulos, cuando se haga referencia a un vértice se refiere a un vértice de un triángulo. La superficie de un objeto es dibujada como una colección de triángulos y el *pipeline* debe transformar esta colección de triángulos de modo que le asigne un color a cada pixel de la pantalla.

La etapa de control de vértices transforma los datos de los triángulos a una manera que el hardware interprete y pone estos datos en el caché de vértices. Después, la etapa de sombreado

[4]DirectX® es una marca registrada de Microsoft Corporation.
[5]Microsoft® es una marca registrada de Microsoft Corporation.

de vértices, transformación y luz transforma a los vértices del caché de vértices y asigna datos por vértice como colores, texturas, entre otros.

La etapa de preparación de triángulos crea ecuaciones que son usadas para interpolar colores y otras para los vértices que toquen otros triángulos. La etapa de creación de retícula (en inglés a esta etapa se le llama *Raster Stage*) se encarga de determinar qué pixeles de la pantalla están contenidos en cada triángulo y para cada uno de estos pixeles interpola los valores necesarios para sombrear, colorear, posicionar y aplicar textura al pixel.

La etapa de sombreado determina el color final de cada pixel. Mientras que la etapa de operación sobre retícula (*Raster Operation*) se encarga de suavizar aquellos objetos que se traslapan o que están muy juntos. Esta etapa también determina los objetos visibles desde un punto de vista determinado y descarta los pixeles ocultos. Finalmente la etapa FBI (por sus siglas en inglés *Frame Buffer Interface*) se encarga de escribir los datos en el búffer de la pantalla.

1.2.2. Creación de *pipelines* programables.

En 2001 NVIDIA creó sus primeras tarjetas que permitían la programación de ciertas etapas del *pipeline*. En particular etapas de sombreado de pixeles y aplicación de texturas. Conforme se empezaron a popularizar operaciones diseñadas por los desarrolladores también se fueron incluyendo unidades especiales para llevar a cabo operaciones que requirieran gran cantidad de aritmética de punto flotante.

1.2.3. La primera generación de GPU compatibles con CUDA. La G80 de NVIDIA.

En 2006 NVIDIA lanzó al mercado la tarjeta gráfica GeForce 8800 la cual transformó las distintas etapas programables del *pipeline* en un arreglo de procesadores unificados. Este arreglo de procesadores unificados permite dividir las etapas en sombreado de vértices, procesamiento geométrico y procesamiento de pixeles.

Las GPU constan de dos partes fundamentales: memoria y un conjunto de (del inglés *Stream Multiprocessor*). Cada cuenta con núcleos de procesamiento (en inglés *Stream Processor*). Lo más importante a notar es que las GPU son un arreglo de , donde cada tiene un número determinado de núcleos o (usaremos núcleo de aquí en adelante). En la arquitectura G80 cada tiene ocho núcleos. Este es el aspecto clave en el escalamiento de las GPU pues para tener más poder de cómputo solo se tienen que agregar más o más núcleos a cada . En la Figura 1.2 se muestra de manera simplificada esta arquitectura de NVIDIA.

Como ya se dijo, un consta de varios núcleos. Estos son los encargados de procesar los datos. Además cuenta con un bloque de memoria llamado *archivo de registros* que es un bloque de memoria que tiene la misma velocidad que los núcleos por lo que es una memoria sin espera. Esta se usa para alojar los datos de registros que están usando los núcleos. Cada tiene un bloque de memoria privado llamado *memoria compartida*. Dicho bloque puede actuar como un caché privado y compartido por el . Cabe señalar que cada tiene un bus separado para

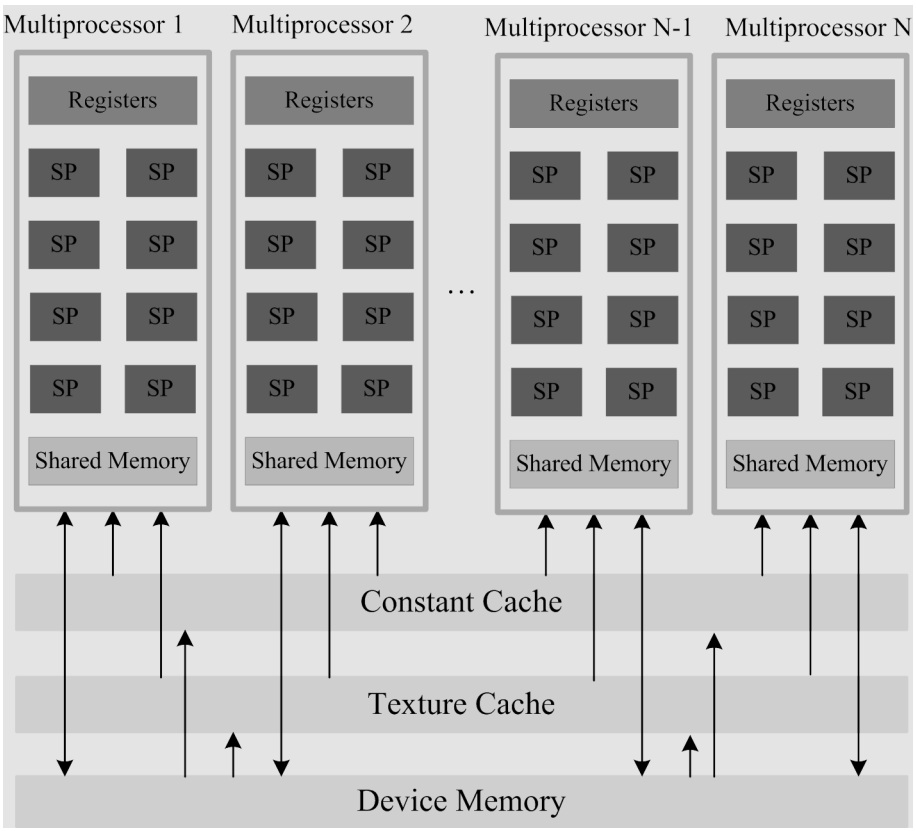

Figura 1.2: Imagen simplificada de la arquitectura G80 de NVIDIA.

acceder a la , a la memoria de textura y a la memoria constante. La memoria constante actúa como una memoria de solo lectura. La memoria de textura se especializa en accesos con ciertos patrones en una aplicación que así lo solicite.

La es GDDR (que son las siglas en inglés para *Graphics Double Data Rate*) que es una versión de alto desempeño de la memoria DDR (*Double Data Rate*) presente en la mayoría de las computadoras personales actuales. El ancho del bus de la memoria puede ser de hasta 512 bits siendo entre 5 y 10 veces más que el encontrado en las memorias para computadoras personales.

Cada tiene al menos dos unidades que en inglés se les llama (Special Purpose Units). Estas unidades implementan a nivel de hardware operaciones útiles como senos y cosenos u operaciones con exponentes.

1.2.4. Capacidades de cómputo.

Con el desarrollo de CUDA se han presentado diversas versiones que el hardware soporta directamente. A estas versiones de hardware para CUDA se les conoce como capacidades de cómputo. Las GPU con chip G80 tienen la primera versión de CUDA. Dado que estas son restricciones de hardware, para cambiar de una versión a otra más reciente es necesario cambiar la tarjeta por completo. Hasta la fecha, en cada generación se ha duplicado el poder

de cómputo y ampliado las capacidades de las tarjetas, así, como el número de componentes de las mismas.

Capacidad 1.0.

Esta fue la primera generación de tarjetas CUDA y su principal defecto de componentes es que no soportan las llamadas operaciones *atómicas* que son operaciones a nivel de hardware son realizadas sin interrupción de hardware o software (estas operaciones se describen con mayor detalle en la sección 4.3.1). Sin embargo, estas tarjetas son consideradas obsoletas y es muy difícil ver alguna funcionando hoy en día.

Capacidad 1.1.

Esta capacidad puede ser encontrada en las series 9000, como la 9800 GTX que fue muy popular. Estas están basadas en la arquitectura G92 que es una mejora a la G80 original.

Un gran cambio en algunos de estos modelos es la ejecución de programas y transferencia de datos de manera simultánea. Esta característica permite utilizar a nivel de hardware la técnica conocida como *doble búffer* que consiste en tener dos búffers reservados y hacer que mientras la GPU procesa uno de ellos, el subsistema DMA llene el otro con los datos. Esto permite que tanto el motor de copiado del CPU como la GPU estén ocupados y que cuando la GPU termine de procesar un grupo de datos, haya ya otros listos para ser procesados.

Capacidad 1.2.

Esta capacidad se encuentra en las series GT200 y NVIDIA duplicó el número de núcleos CUDA comparado con la generación anterior. También NVIDIA incrementó el número de hilos, que se ejecutan al mismo tiempo en un , de 24 a 32. A este grupo de hilos se les conoce como *warp*. Además, se eliminaron algunas restricciones en el uso de la memoria compartida, presentes en generaciones anteriores. Esto provocó que estas tarjetas fueran más fáciles de programar y mejoró considerablemente el desempeño de programas escritos anteriormente con CUDA.

En esta generación se agregaron también algunas operaciones atómicas.

Capacidad 1.3.

Esta capacidad fue desarrollada inmediatamente después de la anterior logrando que casi todas las tarjetas de gama alta de esta generación la tuvieran. La principal mejora radica en que permiten el uso de operaciones de precisión doble al incluir el hardware necesario. Cabe señalar que los gráficos utilizan muchas operaciones de precisión simple pero pocas de precisión doble por lo que el uso de estas operaciones no es muy recomendable pues reducen el desempeño de los programas. Sin embargo, programas que combinen el uso de operaciones

de precisión simple y doble ocupan ambas unidades de hardware por lo que mejoran el desempeño.

1.3. Arquitectura Fermi.

Fermi[6] es el nombre que recibe la arquitectura que tiene capacidades de cómputo 2.x. Los principales cambios de la capacidad 2.0 son los siguientes:

- Introducción de caché L1 desde 16 KB hasta 48 KB.
- Introducción de caché L2 compartido por todos los .
- Dos motores de copia para tarjetas Tesla[7].
- Extensión del tamaño de memoria compartida de 16 KB a 48 KB por .
- Los datos se alinean a 128 bytes.
- El número de bancos de memoria compartida incrementaron de 16 a 32.

A continuación se describen con detalle lo que estos cambios permitieron. Primero, el caché L1 es un tipo de caché presente en cada y es el más rápido que existe. Las capacidades 1.x no tienen caché mas que el de textura y el de memoria constante. Tener un caché le permite al programador crear aplicaciones que funcionen mejor y más rápido en la GPU.

El caché L2 tiene un tamaño de hasta 768 KB en la arquitectura Fermi y es un caché unificado. Es decir, es visto y compartido por todos los . Esto permite una comunicación entre bloques mucho más rápida por medio de las operaciones atómicas. Si lo comparamos con la fuera del chip, el caché L2 es hasta 8 veces más rápido.

Un *stream* es un mecanismo que permite ejecutar procedimientos en el dispositivo a manera de *pipeline*. Los motores de copia doble sirven para ejecutar *streams* utilizando la técnica de doble búffer mencionada anteriormente. De esta manera las operaciones de copiado de memoria se ocultan por la ejecución del procedimiento de otro *stream*. Así se utilizan al mismo tiempo las partes más importantes del dispositivo.

La memoria compartida también cambió drásticamente pues fue combinada con el caché L1. El tamaño del caché completo es de 64 KB; sin embargo, para mantener compatibilidad con programas anteriores siempre se debe reservar al menos 16 KB de memoria compartida, dejando 48 KB para el caché. La arquitectura permite intercambiar estas dos cantidades permitiendo tener memoria compartida de hasta 48 KB lo cual beneficia a muchos programas.

Las alineaciones de datos en la memoria se volvieron más estrictas debido a la introducción de los cachés. Ambos tienen líneas de caché de 128 bytes. Una línea de caché es la mínima cantidad de datos que se pueden leer desde la memoria. Ésto funciona muy bien cuando el programa adquiere localidades consecutivas de la memoria, lo cual es bastante común en

[6]Fermi® es una marca registrada de NVIDIA Corporation.
[7]Tesla® es una marca registrada de NVIDIA Corporation.

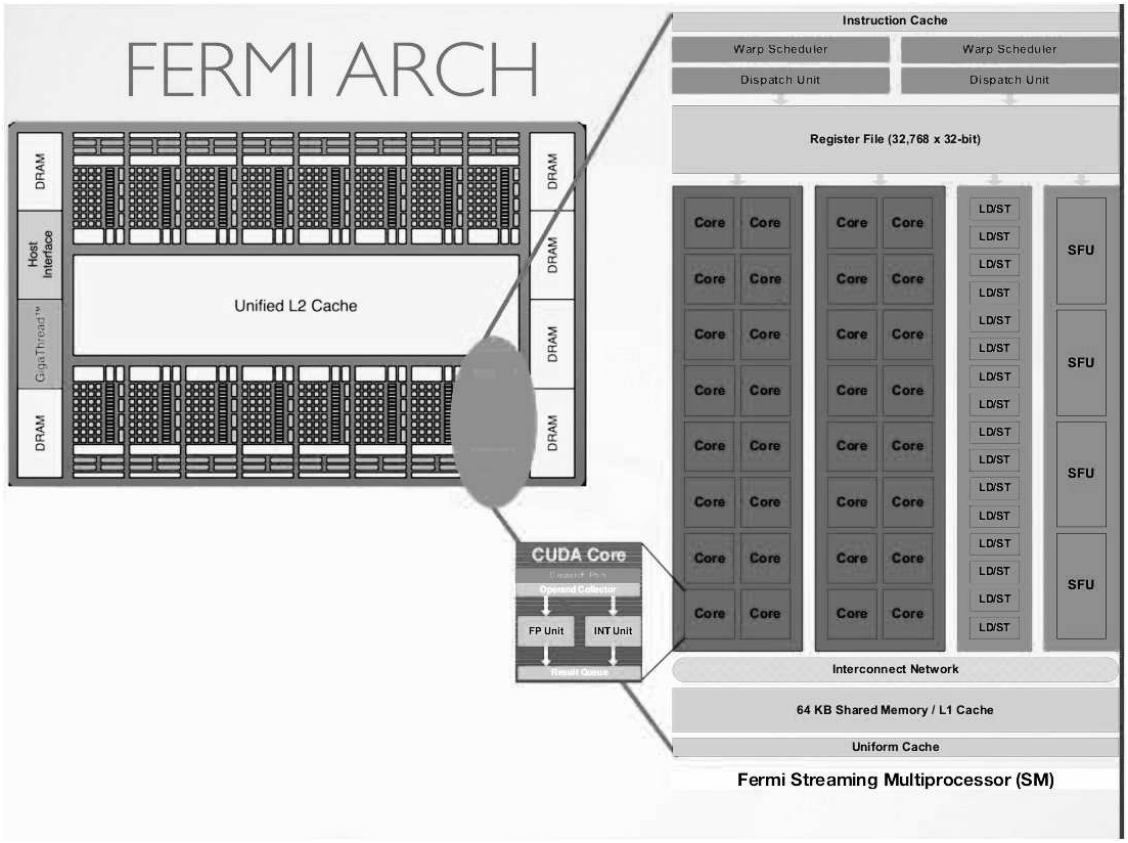

Figura 1.3: Imagen de la arquitectura Fermi de NVIDIA.

CUDA. Si el programa tiene un patrón de acceso a memoria disperso se puede deshabilitar esta función y utilizar el alineamiento de 32 bits de las generaciones anteriores.

Finalmente, el último de los mayores cambios consiste en de incrementar el número de bancos de memoria de 16 a 32. Esto beneficia al número actual de hilos en un *warp* que es 32, pues cada hilo puede acceder a un banco distinto en memoria compartida sin ocasionar conflictos.

En la Figura 1.3 se muestra la arquitectura Fermi. En ella se aprecia la organización de toda la GPU, así como el contenido de cada y el hardware incluido en cada núcleo ().

Capacidad 2.1.

Esta capacidad está presente en tarjetas de la generación Fermi enfocadas al mercado de videojuegos y presenta algunos cambios como los siguientes:

- 48 núcleos CUDA por , en lugar de 32.
- 8 unidades de propósito especial de precisión simple por , en vez de 4.
- Doble despachador de *warps* en lugar de uno sencillo.

Esta capacidad sacrifica hardware para cálculos con precisión doble pero se enfoca en incrementar el número de núcleos CUDA. Para procesos de precisión simple y con aritmética entera este es un buen trueque. Como el mercado de esta arquitectura son los videojuegos, estos utilizan muchos cálculos de precisión simple pero pocos de precisión doble.

En la capacidad 2.0 se requieren dos ciclos de reloj para asignar instrucciones a todos los hilos de un *warp*. En la capacidad 2.1 en lugar de despacharse dos instrucciones cada dos ciclos se despachan cuatro. Otro cambio es el incremento en el número de núcleos por . Ahora en lugar de dos líneas de 16 núcleos se tienen tres dando un total de 48 núcleos por .

Esta arquitectura fue la primera con la que se trabajaron los programas presentados en los casos de estudio; sin embargo, para el tiempo de desarrollo de los mismos surgió la nueva arquitectura que es presentada en la siguiente sección.

1.4. Arquitectura Kepler.

En 2012 NVIDIA lanzó su nueva arquitectura de GPU a la que le llamó Kepler[8]. Esta arquitectura está enfocada en la eficiencia energética pues pretende mejorar el desempeño por vatio consumido. Para esto NVIDIA creó un nuevo llamado SMX que presenta una arquitectura un tanto diferente a las versiones anteriores y que mejora el reloj interno para mejorar la eficiencia energética, dado que el número de núcleos se incrementó considerablemente esto logra un mejor desempeño energético.

El primer cambio en los son las unidades de despacho de instrucciones. En Kepler se tienen cuatro calendarizadores de *warps* y cada calendarizador provee dos instrucciones por *warp* en cada ciclo de reloj. Más aún, aunque tanto Fermi como Kepler comparten muchas unidades de hardware, los calendarizadores de Fermi contenían unidades complejas para prevenir fallos de instrucciones, pero los diseñadores de NVIDIA notaron que estos fallos se pueden calcular en tiempo de compilación así que los calendarizadores de Kepler son más sencillos logrando que utilicen menos energía.

Otro cambio es el incremento de núcleos por , pues estos llegan a los 192 núcleos CUDA por , acumulando un total de 1536 en todo el dispositivo a comparación de Fermi que alcanzaba los 512 núcleos CUDA. Sin embargo, se decrementó el número de pues en la GTX 580 de arquitectura Fermi se tenían 16 mientras que en la GTX 680 con arquitectura Kepler se tienen solo 8 (recordando que ahora son SMX).

También se incrementó el caché L2 a 512 KB y también el ancho de banda un 73 %. Así mismo, se mejoraron notablemente las operaciones atómicas pues ahora se puede realizar una operación atómica por ciclo de reloj cuando la dirección de la operación es compartida, esto mejora 9 veces el rendimiento anterior.

En esta arquitectura también se mejoró la memoria DRAM del dispositivo, creando en particular para la tarjeta GeForce GTX 680, una memoria GDDR5 con una velocidad de 192 GigaBytes por segundo.

[8]Kepler® es una marca registrada de NVIDIA Corporation.

1.4.1. Capacidad 3.5.

El chip GK110 es una versión mejorada de la arquitectura Kepler y tiene la capacidad de cómputo 3.5. Este chip está presente en las tarjetas de gama alta de la actual serie GeForce, como la GeForce Titan[9].

Las mejoras radican en los SMX pues los de este nuevo chip cuentan con 192 núcleos CUDA de precisión simple y cada núcleo tiene un *pipeline* completo para aritmética entera y de precisión simple.

Otra ventaja es que cada SMX tiene un calendarizador (en inglés *scheduler*) cuádruple de *warps* y ocho unidades de despacho de instrucciones, permitiendo que cuatro hilos sean ejecutados y calendarizados concurrentemente. Este SMX selecciona cuatro *warps* y dos instrucciones independientes por *warp* pueden ser despachadas cada ciclo.

Otro cambio en esta capacidad es la cantidad de registros que un hilo puede usar. Antes solo se podían usar hasta 63 registros por hilo, ahora se pueden usar hasta 255. Además de que el número de registros por SMX se incrementó a 65536. Esto beneficia a programas que utilicen muchos registros y que antes presentaban problemas por no tener registros disponibles.

Tal y como la versión anterior, en esta las operaciones atómicas se mejoraron y se agregaron más operaciones atómicas.

La mejora más atractiva para los programadores es el *paralelismo dinámico*. Esto básicamente significa que un proceso en la GPU puede lanzar otros procesos de manera dinámica y esperar los resultados para continuar trabajando. Anteriormente, la única manera de lanzar procesos era a través del CPU como se describe en el próximo capítulo.

[9]Titan® es una marca registrada de NVIDIA Corporation.

Capítulo 2

Lenguaje de programación CUDA C.

En el capítulo anterior se relató la historia de las GPU y al final se mostró la arquitectura más reciente en las tarjetas NVIDIA. En este capítulo se expone una manera de utilizar la tarjeta gráfica como dispositivo de cómputo general. Para ello se describe una extensión de un lenguaje de programación de alto nivel ya existente.

En la primera sección se presentan los conceptos más generales de CUDA C. En la segunda sección se analizan distintos tipos de memoria y sus usos para crear aplicaciones paralelas. En la última sección se considera un principio fundamental en las aplicaciones paralelas con CUDA donde se pretende mejorar el rendimiento de las mismas utilizando más operaciones para reducir la latencia de los accesos a memoria y también se describe un problema inherente a la arquitectura CUDA al momento de lanzar hilos que tengan que ejecutar código distinto.

2.1. CUDA C como extensión del lenguaje C.

El lenguaje de programación *C* creado por Dennis Ritchie a principios de los años 70 sigue siendo uno de los más utilizados a nivel mundial en distintos ámbitos de software, en particular para desarrollo de sistemas operativos, controladores de dispositivos y aplicaciones que requieran un uso eficiente de memoria y de hardware.

NVIDIA decidió utilizar este lenguaje como base para crear su plataforma de programación de dispositivos[1]. En lugar de crear un lenguaje de programación nuevo y con características paralelas, NVIDIA decidió extender el lenguaje C por medio de una serie de funciones y extensiones a la sintaxis como palabras reservadas y operadores. Algunas de estas características se describen a lo largo de este capítulo. Cabe señalar que lo aquí presentado es solo una descripción para poder entender el código de capítulos posteriores y no es un tutorial

[1]NVIDIA también creo una plataforma de programación de dispositivos basada en el lenguaje de programación FORTRAN.

Código 2.1: *Kernel* que imprime "hola mundo" escrito en CUDA C.

```
__global__ void kernel(){
        printf(``hola mundo\n'');
}
```

ni material suficiente para aprender CUDA C. Para dicho propósito se recomienda revisar la literatura: [8, 13, 22].

Un programa que tenga código para ser ejecutado en un dispositivo debe tener la extensión *.cu* y primero debe ser compilado con el compilador de NVIDIA *nvcc*. Esta herramienta determina qué partes del código son propias de CUDA C y deben ser ejecutadas en el dispositivo y qué partes deben serlo en el anfitrión. Por anfitrión entendemos al CPU.

Entonces, para poder ejecutar código en un dispositivo este se debe invocar desde una función de código anfitrión, es decir, desde una función en C (no necesariamente la función *main* de C). El compilador de NVIDIA detecta todas las funciones, palabras, variables y macros que sean propias de CUDA C y las procesa; después, invoca al compilador de C existente en la máquina donde se esté compilando para que el código del lenguaje C se compile también. Si la compilación tiene éxito se crea un archivo ejecutable con el código proveniente de C junto con las rutinas propias para poder utilizar el dispositivo durante la ejecución y este ejecute el código que le fue programado.

2.1.1. Funciones principales de dispositivo.

Para poder ejecutar código en el dispositivo se debe crear una función con notación propia de CUDA C y esta debe ser invocada desde otra función que se ejecute en el anfitrión. Para distinguir entre este tipo de funciones, a las funciones de dispositivo que se quieran ejecutar desde una función de código anfitrión se utiliza el modificador `__global__` antes su declaración. A las funciones con este modificador se les conoce como *kernels*[2].

Este modificador indica que dicha función ejecutará su código en un dispositivo y puede ser invocada desde código anfitrión. En el Código 2.1 se presenta un pequeño ejemplo donde el *kernel* solo imprime un "hola mundo".

Una vez creada la función, para ser ejecutada en el dispositivo se debe invocarla desde código anfitrión, para ello se necesita una sintaxis propia de CUDA C. La razón de esta sintaxis es porque la función que se ejecutará en el dispositivo será ejecutada por los núcleos CUDA (los mismos núcleos descritos en el capítulo anterior).

En el lenguaje CUDA C existe una jerarquía de organización para simular la organización de los componentes de hardware. La unidad más pequeña en esta organización es un *hilo* (en inglés *thread*). Un hilo CUDA es simplemente una entidad administrativa que ejecuta una

[2] *Kernel* en alemán significa núcleo y esta palabra hace referencia, en la literatura en inglés, a que estas funciones son las funciones iniciales de código en GPU. A los componentes principales de un sistema operativo también se les conoce en conjunto con este nombre.

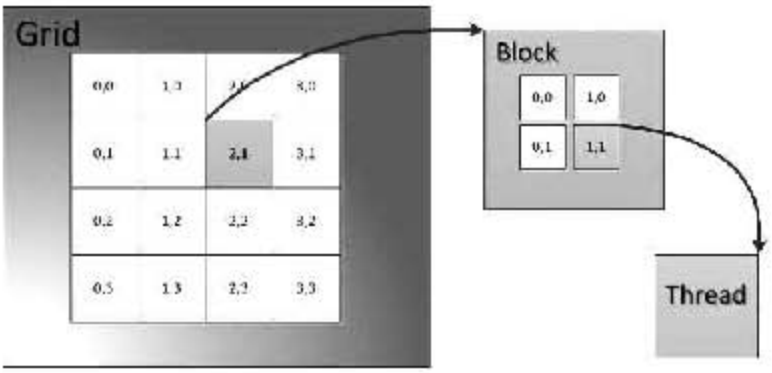

Figura 2.1: Imagen de la jerarquía de hilos en CUDA C: Cada hilo pertenece a un bloque y cada bloque pertenece a una malla. Un *kernel* lanza una malla compuesta por al menos un bloque y cada bloque se compone de al menos un hilo.

función marcada con el modificador `__global__`. Sin embargo todo hilo debe pertenecer a un bloque.

Un *bloque* (en inglés *block*) es la segunda unidad de organización. Para que se ejecute código en el dispositivo se necesita al menos un bloque que contenga al menos un hilo. Al conjunto de todos los bloques que se lanzan en paralelo para ejecutar una función dada se le llama *malla* (en inglés *grid*). A esta organización se le llama *jerarquía de hilos* en CUDA C y se muestra en la Figura 2.1.

Cuando se lanza un *kernel* se debe establecer el número de bloques que se lanzarán y el número de hilos que tendrá cada bloque. Esta jerarquía es una abstracción de la organización física en la arquitectura. Un bloque es asignado a un y aunque en un mismo se pueden alojar varios bloques, solo uno está activo en todo momento. Los hilos representan a los SP. Es decir, las instrucciones que cada hilo ejecuta son ejecutadas por un núcleo CUDA. Como usualmente se lanzan más bloques que disponibles, entonces la tarjeta se encarga de administrar qué bloque es asignado a qué y esto ocurre también con los hilos, pues se pueden lanzar más hilos por bloque que núcleos disponibles en el ; sin embargo, en el caso de los hilos,estos se administran por *warps*, es decir, el calendarizador de un calendariza *warps* completos y nunca hilos individuales. Esto es un factor a tener en cuenta al momento de programar las aplicaciones.

Para invocar una función desde el dispositivo se utiliza la notación `<<<x,y>>>` en donde x representa el número de bloques deseado mientras que y representa el número de hilos que tendrá cada bloque. Aunque esta notación acepta otros argumentos, dichos argumentos no son necesarios y representan cosas muy específicas.

Cabe señalar que hay restricciones de hardware para el número máximo de hilos por bloque al igual que al número total de bloques a lanzar. En las tarjetas más modernas el máximo número de hilos por bloque asciende a 2048, mientras que el número máximo de bloques a lanzar asciende a 65535.

Por ejemplo, para ejecutar la función creada anteriormente, digamos con 5 bloques conteniendo 10 hilos cada uno se debe escribir dentro de una función desde el anfitrión (en este caso será desde el *main*) como se muestra en el Código 2.2.

Código 2.2: Invocación de un *kernel* desde el anfitrión.

```
int main(){
        kernel<<<5,10>>>();
        return 0;
}
```

2.1.2. Funciones globales y locales en dispositivos.

Ya se describió cómo crear una función global que se ejecutará en el dispositivo y cómo invocarla desde código anfitrión especificando tanto la cantidad de bloques como el número de hilos por bloque que la ejecutarán. Sin embargo, si los hilos deben hacer una tarea en repetidas ocasiones dentro de un *kernel*, CUDA C provee un mecanismo para llevar a cabo esta tarea o tareas más generales que sean resueltas mediante el concepto abstracto de *función local*.

Para crear una función local en el dispositivo, la cual ejecutarán todos los hilos cuando sea invocada desde una función principal, se debe añadir a la definición de la función el modificador `__device__`. Las funciones de dispositivo entonces, son ejecutadas desde el mismo pero siempre dentro del cuerpo de una función global o dentro del cuerpo de otra función de dispositivo. Las funciones de dispositivo no se pueden ejecutar desde código anfitrión ni necesitan la notación para indicar el número de hilos y de bloques que las ejecutarán.

El compilador interpreta una función con el modificador `__device__` como una función exclusivamente de dispositivo; además, si una función no tiene un modificador global o de dispositivo, será ignorada por el compilador de CUDA C y será compilada exclusivamente por el compilador de C, por lo que invocarla desde código de dispositivo causará un error de compilación.

Si se quiere ejecutar una función de C en el dispositivo o se tienen funciones que se requieren usar tanto en código anfitrión como en código de dispositivo, CUDA C provee otro modificador para hacer que una función de dispositivo pueda ser utilizada como una función de código anfitrión, lo que resuelve los problemas iniciales. Así, si ya se tiene una función escrita en C, para utilizarla en el dispositivo se le agrega el modificador `__device__` al inicio de su declaración y para que a su vez sea posible utilizarla en código anfitrión se le añade el modificador `__host__`. Es decir, para que una función se pueda ejecutar tanto en el anfitrión como en el dispositivo se agregan ambos modificadores.

Por ejemplo, se podría tener una función tanto en dispositivo como en el anfitrión que calculara la norma al cuadrado de dos vectores recibiendo cuatro números como argumentos como se muestra en el Código 2.3.

2.1.3. Manejo dinámico de memoria del dispositivo.

El lenguaje C es conocido por permitir al programador el manejo dinámico de memoria, es decir, permite asignar espacios contiguos de memoria a petición explícita del programador

Código 2.3: Declaración de funciones para ser ejecutadas tanto en dispositivo como en el anfitrión.

```
//(x1,y1) es el primer vector y (x2,y2) el segundo
__host__ __device__ float normaAlCuadrado(
float x1, float y1, float x2, float y2){
        return ((x2 - x1) * (x2 - x1)) + ((y2 - y1)*(y2 - y1));
}
```

Código 2.4: Asignación y liberación de memoria para el dispositivo.

```
int *d_pointer;
cudaMalloc(&d_pointer,1000 * sizeof(int));
//se ejecuta un kernel o se copia algo a ese buffer
//...
cudaFree(d_pointer);
```

y que estos sean liberados después mediante otra petición. Dichas funciones presentes en C son `malloc` para asignar la memoria y `free` para liberar memoria previamente asignada con la primera función.

Este modelo de manejo dinámico de memoria está presente en CUDA C. En el lenguaje de NVIDIA existen funciones que permiten asignar y liberar memoria en el dispositivo y deben ser ejecutadas por el anfitrión.

CUDA C presenta ciertos estándares en el nombramiento de funciones particulares que tienen un parecido en su funcionamiento con funciones ya existentes en C por lo que la función para asignar memoria en el dispositivo se llama `cudaMalloc()` dicha función toma como argumentos la dirección de un apuntador que será la dirección del inicio del búffer, este argumento debe tener tipo `void**`. Toma también un número no negativo que será el número de bytes a asignar en el dispositivo y asigna ese número de bytes empezando en la dirección del primer argumento. Hay que observar que dicho apuntador tiene el mismo tipo que un apuntador de código anfitrión pero hay que tener en cuenta que su valor (la dirección a la que apunta) forma parte de una dirección de memoria del dispositivo, que forma un espacio físico de direcciones independientes y distinto al mantenido por el anfitrión.

Para liberar memoria de un dispositivo desde el anfitrión se utiliza la función `cudaFree()` cuyo argumento es un apuntador que haya sido pasado como argumento a `cudaMalloc()`.

De esta manera, CUDA C permite un manejo de memoria muy similar al proporcionado por el lenguaje C, por lo que un desarrollador que conozca el lenguaje se familiarizará con CUDA C en poco tiempo. En el Código 2.4 se muestra un ejemplo del uso de estas funciones en donde se reserva un búffer en el dispositivo de tamaño 1000 enteros (`int`).

Código 2.5: Copiado de memoria hacia y desde el dispositivo.

```
cudaMemcpy(dev_a,a, n * sizeof(int), cudaMemcpyHostToDevice);
//se procesan los datos en el dispositivo..
//...
cudaMemcpy(a, dev_a,n * sizeof(int), cudaMemcpyDeviceToHost);
```

2.1.4. Transferencia de datos entre anfitrión y dispositivo.

Los programas, en su descripción más general, tienen una serie de datos de entrada. Los programas que se ejecutan en un CPU toman estos datos de diversos medios: leyendo contenido de archivos, interpretando señales externas provistas por usuarios como apretar teclas, botones o secciones de la pantalla o incluso movimientos del ratón. Sin embargo, un programa que se ejecuta en un dispositivo no tiene acceso a estas señales ni manera de interpretarlas (en el caso del CPU es el sistema operativo el que interpreta las señales y mediante llamadas al sistema el programa puede tener acceso a ellas) más aún, el dispositivo tampoco tiene acceso a archivos o búffers del sistema operativo anfitrión. Pero el dispositivo sí tiene acceso a su propia memoria.

Es por medio de la memoria que un programa en CUDA recibe sus datos de entrada y para ello se deben almacenar dichos datos en la del dispositivo antes de que se empiece a ejecutar el programa. Por lo tanto es tarea del código ejecutado por el CPU llevar a cabo este almacenamiento.

Como los espacios de direcciones de memoria son distintos e independientes el anfitrión no puede escribir de manera directa datos en la memoria del dispositivo sino que debe utilizar ciertas funciones que permiten copiar datos entre las distintas memorias.

Cualquier función de la familia `cudaMemcpy()` puede realizar estas copias, la función `cudaMemcpy()` recibe, en el siguiente orden: un apuntador que señale a la dirección destino de la copia, un apuntador que señale a la dirección fuente de la copia, el número de bytes a ser copiados y finalmente el sentido de la copia. Este sentido de copia está dado por dos palabras reservadas propias de CUDA C. La primera, `cudaMemcpyHostToDevice`, permite copiar el contenido desde el anfitrión hacía el dispositivo. La segunda es análoga y permite la copia en el sentido inverso, `cudaMemcpyDeviceToHost` copia el contenido del segundo apuntador en el primero, el segundo debe ser un apuntador de dispositivo mientras que el primero debe ser uno en el anfitrión.

Por lo general, se utilizan ambas direcciones como complementos ya que primero se copia desde el anfitrión al dispositivo, luego se procesan esos datos mediante una serie de funciones globales en el dispositivo y, finalmente, se regresan los datos al anfitrión para ser guardados, mostrados o procesados de nuevo pero ahora por el CPU como se muestra en Código 2.5.

También `cudaMalloc` y `cudaFree` deben usarse como complementos, pues al no haber un sistema operativo que administre la memoria en el dispositivo, si dicha memoria no se libera puede llegar a causar errores en programas futuros. Cabe señalar que `cudaMalloc` solo asigna el espacio, no lo inicializa; así que, el programador debe de tener esto en cuenta al momento

Código 2.6: Declaración de un búffer de memoria compartida.

```
__shared__ int *var;
```

de intentar acceder a esas direcciones de memoria en el dispositivo.

2.2. Tipos de memoria.

Existen varios tipos de memoria en CUDA C. En esta sección se presentan los más usados y sus principales características.

2.2.1. Memoria global.

Esta es la memoria más general y la más usada. Es asignada y liberada desde el anfitrión con las funciones mencionadas en la sección anterior.

La puede ser manipulada mediante asignaciones básicas, apuntadores y puede ser referenciada y desreferenciada mediante los operadores que provee el lenguaje C. Además, si se trata de acceder a alguna dirección no asignada mediante `cudaMalloc` el comportamiento no está definido y lo más probable es que ocurra un error en tiempo de ejecución tal y como sucede con la memoria no asignada en el anfitrión.

2.2.2. Memoria compartida.

La memoria compartida reside en una sección especial del dispositivo tal y como se mencionó en el capítulo anterior. Para hacer uso de esta memoria en CUDA C es necesario declarar una variable como compartida. Para ello se hace uso del modificador `__shared__` antes del tipo de la variable como se muestra en el Código 2.6.

CUDA C trata a las variables compartidas de manera distinta a las variables normales. Para declarar una variable compartida, la declaración debe hacerse dentro de una función de CUDA C ya sea global[3] o de dispositivo. El efecto que tiene esta declaración es que CUDA C crea una copia diferente de esta variable para *cada* bloque que ejecute la función actual y *todos* los hilos del bloque comparten el estado de la variable durante su ejecución.

Esta memoria resulta muy útil pues es mucho más eficiente. Recordemos que esta memoria reside directamente en el chip de la GPU y no en DRAM como la , además provee un mecanismo de comunicación entre hilos del mismo bloque. Sin embargo, recordemos que esta memoria tiene un tamaño muy limitado por a diferencia de la .

Al hablar de comunicación entre hilos del mismo bloque se necesita mencionar una manera de sincronizarlos. Esto se hace mediante la instrucción nativa de CUDA C `__syncthreads()`.

[3]Usualmente se declara dentro de este tipo de funciones y al inicio de las mismas.

Esta instrucción provee un mecanismo de sincronización de barrera entre hilos del mismo bloque, por lo que el primer hilo que la ejecuta detendrá la ejecución de instrucciones sucesivas hasta que todos los hilos en ejecución del bloque hayan terminado de ejecutar esta instrucción. Con este mecanismo de sincronización y el uso de la memoria compartida se puede hacer un uso muy eficiente de cómputo paralelo con baja latencia de memoria a nivel de bloques CUDA.

2.2.3. Memoria constante.

La memoria constante, como su nombre lo indica, es un tipo especial de memoria de *solo lectura*. Esto es, que una vez asignada no puede ser reasignada. Esta memoria también se encuentra presente dentro del chip y en cantidades limitadas tal como se mostró en el capítulo anterior. La memoria constante es efectiva cuando se quiere trabajar con un conjunto de datos al cual no se le quiere hacer modificaciones para producir la salida. En otras palabras cuando el conjunto de datos de entrada solo se utiliza para *leer* los datos y la salida generada es en otro conjunto independiente de datos.

Como hay un caché en el chip dedicado a esta memoria, la latencia en particular de lectura es mucho menor que la de la . Por lo que si se usa exhaustivamente se logrará mejor rendimiento que si se leyeran los datos de las otras memorias.

Para declarar una variable dentro del espacio de memoria constante se debe usar el modificador `__constant__`. El compilador reserva espacio dentro de la memoria constante para las variables que tengan dicho modificador; sin embargo, el espacio declarado debe tener un espacio conocido en tiempo de compilación. Es decir, en caso de que se declare un arreglo dentro de la memoria constante, dicho arreglo debe tener tamaño constante (algo parecido a lo que se necesita en C++ cuando se declara una variable constante). Otro punto que hay que especificar acerca de la declaración de memoria constante es que esta se declara en código anfitrión como se declaran los apuntadores para la a diferencia de la memoria compartida.

Otro cambio en el uso de esta memoria con respecto a la global es la asignación propia del espacio constante. En la se utiliza la función `cudaMemcpy()` y se utiliza un indicador para decir que se esta copiando memoria del anfitrión hacia el dispositivo. La memoria constante también se asigna desde el anfitrión mediante la función `cudaMemcpyToSymbol()`. Esta función toma dos apuntadores, el primero es el destino de la copia, el segundo es el origen y toma también, como tercer argumento, el número de bytes a ser copiados. A diferencia de la primera función, esta copia a memoria constante desde el anfitrión. Como la memoria constante reside en el chip, no se necesita liberar y tampoco puede ser copiada de vuelta al anfitrión por lo que esta función no necesita saber la dirección de copiado pues la dirección siempre es desde el anfitrión hacia la memoria constante del dispositivo. En el Código 2.7 presenta un ejemplo.

Código 2.7: Copiado de memoria constante desde el anfitrión.

```
__constant__ int * dev_array;
// int *array contiene los datos a copiar en el dispositivo
//y contiene N enteros.
cudaMemcpyToSymbol(dev_array, array, sizeof(int)* N);
//se procesan los datos.
```

Nombre	Declaración	Residencia	Alcance	Tiempo de vida.
Variables locales	`int var`	registros	hilo	*kernel*
Memoria compartida	`__shared__ int var`	memoria compartida	bloque	*kernel*
Memoria constante	`__constant__ int var`	memoria constante	malla	aplicación
Memoria global	`int var`		malla	aplicación

Cuadro 2.1: Tabla comparativa de los tipos de memoria en CUDA C.

2.2.4. Uso actual de los distintos tipos de memorias.

Además de las memorias mencionadas existen otro tipo de memorias como la memoria de textura, la memoria fija en el anfitrión[4] o la memoria sin necesidad de copia[5]. Sin embargo, dichas memorias son útiles solo en ciertas aplicaciones no de una manera general como las presentadas anteriormente.

Los distintos tipos de memoria fueron creados para mejorar el rendimiento de las aplicaciones y tratar de reducir el mayor cuello de botella en rendimiento de las mismas: el ancho de banda y la latencia de la memoria. Sin embargo, como se mostró en el capítulo de la arquitectura de las GPU, los modelos actuales tienen hasta dos niveles de memoria caché por por lo que varias memorias ya no mejoran tanto el rendimiento como lo hacían en los modelos anteriores cuando no había memoria caché.

En la mayoría de las aplicaciones las memorias usadas son la y la memoria compartida, dado que con la incorporación del caché, la memoria constante y la memoria de textura ya no producen tanta mejora en el rendimiento. Estas dos memorias no serán utilizadas en el trabajo pues aunque se puede utilizar la memoria constante; para hacerlo se necesita, como ya se describió, añadir sintaxis e instrucciones especiales y con la utilización del caché simplemente se deja que el compilador y la unidad de memoria del dispositivo hagan las optimizaciones pertinentes.

En el Cuadro 2.1 se presenta una tabla comparativa de la , la memoria compartida y la memoria constante incluyendo también las variables locales en un *kernel*. En la tabla se comparan: alcances, en qué parte física del dispositivo residen, el tiempo de vida de las mismas y la declaración en código CUDA C.

[4]En inglés *pinned memory*.
[5]En inglés *zero-copy memory*.

2.3. Detalles de optimización de programas en CUDA C.

En esta sección se presentan algunos detalles a tener en cuenta para realizar programas eficientes en CUDA C.

2.3.1. Ocupación del dispositivo.

Conforme han evolucionado los procesadores su velocidad ha aumentado considerablemente con respecto a la velocidad de la memoria. Esto ocasiona que el procesador se quede detenido por varias ciclos de reloj mientras espera que se complete una operación de memoria. A la magnitud de este retraso se le conoce como *latencia de memoria*.

Se han ideado dos medidas para atacar este problema. La primera que se adoptó fue la creación de una jerarquía de cachés de diverso tamaño. Esta medida reduce el problema de la latencia debido a que los datos usados más frecuentemente se almacenan en la memoria caché que se accede mucho más rápido que la memoria RAM.

La otra medida, creada más recientemente, consiste en tener en un mismo procesador precargados varios hilos de ejecución. De modo que si uno es detenido porque espera que se complete una operación de memoria, los otros hilos precargados pueden continuar su ejecución. Esta técnica es aprovechada, por ejemplo, por Intel con la tecnología *Hyper-threading*, [1].

El subsistema de memoria de las GPU está diseñado para maximizar las operaciones de memoria realizadas por unidad de tiempo (en inglés a esto se le conoce como *throughput*) sin optimizar la latencia. Esto implica que la latencia en la memoria de las GPU es alta. Debido a esto, para que un programa en CUDA tenga un buen desempeño, este debe mantener ocupados a los multiprocesadores tanto como sea posible. Un concepto importante para llevar a cabo lo anterior es la *ocupación*.

Las instrucciones que ejecutan los hilos son ejecutadas secuencialmente en CUDA y como resultado, ejecutar otros *warps* cuando un *warp* está detenido o pausado es la única forma de esconder las latencias y mantener el hardware ocupado. La *ocupación* es una medida que relaciona el número de *warps* activos en un multiprocesador con el máximo número de *warps* posibles. A mayor número activo de *warps* mayor ocupación tiene el dispositivo.

Una alta ocupación no siempre resulta en un mejor desempeño. Hay un momento en que más ocupación no mejora el desempeño. Sin embargo, una baja ocupación disminuye la posibilidad para ocultar las latencias de la memoria, lo que implica un pobre desempeño.

Uno de los muchos factores que determinan la ocupación es la disponibilidad de registros. El uso de registros permite a los hilos tener variables locales en memoria con acceso de bajo costo para que tenga baja latencia. Sin embargo, el conjunto de registros disponibles es limitado y todos los hilos en el multiprocesador deben compartirlo. Los registros deben estar asignados para todo un bloque al mismo tiempo. Así que, si los hilos de un bloque usan muchos registros, entonces puede que no queden registros disponibles para insertar a otro bloque, por lo que la ocupación del multiprocesador disminuye.

El número de registros disponibles, el máximo número de hilos silmultáneos en un multiprocesador y la precisión en la asignación de registros varían de acuerdo a la capacidad de cómputo de los dispositivos. Dados estos detalles en la asignación de registros y ya que la memoria compartida también está dividida entre los bloques residentes en el multiprocesador, la relación exacta entre el uso de registros y la ocupación puede ser difícil de determinar.

Para calcular la ocupación NVIDIA provee una "calculadora de ocupación" en forma de una hoja de cálculo donde se le introducen datos acerca del *kernel* y otras cuestiones técnicas para que se determine la ocupación del dispositivo Además de esta calculadora, la ocupación puede obtenerse utilizando la "medida de Ocupación Lograda" del perfilador visual de NVIDIA.

Aunque en la arquitectura se pueden lanzar un número muy grande de hilos y bloques a la vez, este número es finito y como ya se dijo puede afectar la ocupación del dispositivo, por lo que se plantea la posibilidad de tener menos hilos que elementos a procesar (lo cual es altamente posible con entradas muy grandes), por lo que se define un ciclo `while` que cada hilo ejecutara y simplemente, en cada iteración, toma los elementos a procesar y una vez procesados incrementa el índice del elemento a procesar por el número total de hilos en ejecución. A esto se le conoce como zancada (o *stride*) y es una técnica muy común en CUDA C.

La técnica de usar un *stride* permite que hilos del mismo *warp* procesen elementos cercanos permitiendo un mejor uso del caché y de la memoria. Además esta técnica también logra que se puedan procesar los elementos sin importar el número de hilos disponibles.

2.3.2. Divergencia de hilos.

Otro detalle que hay que tener en cuenta al diseñar algoritmos en CUDA C es la llamada *divergencia de hilos*. Cuando se lanza un *kernel*, se lanza con un número de hilos y de bloques. Cada uno de estos bloques está formado por uno o más *warps* (recordemos que un *warp* es la manera en que el hardware organiza a los hilos para asignarlos a un núcleo dentro de un). Los hilos de un *warp* ejecutan sus instrucciones de manera secuencial, es decir, hay una sincronización implícita entre ellos. La *divergencia* ocurre cuando los hilos ejecutan instrucciones diferentes.

Cuando hay divergencia de hilos en un *warp*, los hilos que ejecutan instrucciones diferentes lo hacen de manera secuencial. Esto lleva a que un *warp* tarde más en terminar su ejecución pues si hay dos posibles instrucciones (por ejemplo un `if-else`) entonces los hilos que ejecuten la primera deben quedarse detenidos (en inglés *stall*) hasta que los hilos que ejecuten la segunda posible instrucción terminen. Si en algún punto de código la divergencia lleva a todos los hilos a ejecutar de nuevo las mismas instrucciones entonces la sincronización se restablece.

La divergencia de hilos dentro del mismo *warp* conlleva a una pérdida de rendimiento sustancial y siempre se debe tratar de evitar o sino se puede, por lo menos se debe minimizar la sección de código divergente.

Además, los hilos del mismo *warp* no solo deben realizar el mismo trabajo sino que deben realizar trabajo, es decir, cuando solo algunos hilos del *warp* realizan instrucciones y los otros

no realizan trabajo ya sea porque así está el código (los otros cumplen cierta condición que los lleva a no realizar instrucciones) o porque ya terminaron su trabajo entonces también se está perdiendo rendimiento.

El hecho de que las instrucciones a nivel de *warp* se efectúen en paralelo siempre se debe tener en consideración a la hora de diseñar algoritmos en CUDA C.

Capítulo 3

Paradigma *Divide y vencerás.*

En este capítulo se introduce el paradigma *divide y vencerás*. En la primera sección se escribe acerca del paradigma en sí. En la segunda sección se presentan algunos ejemplos y usos del paradigma. Con esto se pretende dar a conocer la diversidad de ejemplos para los que el paradigma funciona. En este capítulo se omiten los algoritmos usados como casos de estudio pues cada uno de estos se expone con mayor detalle en su capítulo correspondiente.

En la tercera sección se describen ventajas y desventajas de este paradigma de diseño de algoritmos. Por último, en la cuarta sección se plantea una manera de rediseñar los algoritmos *divide y vencerás*, esta otra forma es la que se utilizará en los casos de estudio debido a la arquitectura de la plataforma utilizada.

3.1. *Divide y vencerás.*

Divide y vencerás se refiere a un paradigma para diseñar algoritmos en donde el problema a resolver se divide en subproblemas más sencillos. Después, estos subproblemas se resuelven recursivamente y finalmente se combinan estas soluciones para crear la solución del problema original.

A partir de esta descripción del paradigma se pueden ver tres pasos claros para crear un algoritmo *divide y vencerás*:

Dividir el problema en un número de subproblemas que son ejemplares más pequeños del mismo problema. Más pequeños significa que son más sencillos de resolver (esto en general implica que el conjunto de trabajo es, en efecto, menor).

Vencer los subproblemas resolviéndolos recursivamente. Sin embargo, si los subproblemas son lo suficientemente pequeños basta con que se resuelvan de manera directa.

Combinar las soluciones de los subproblemas para crear la solución del problema original.

Cuando se necesita resolver subproblemas que son muy grandes (es decir, que no se pueden solucionar de manera directa) se dice que estos subproblemas son un *caso recursivo* del problema original. Por otro lado, cuando los subproblemas son pequeños y es posible, y de

hecho más factible resolverlos de manera directa, se dice que estos subproblemas son un *caso base* del problema original.

3.1.1. Árbol de recursión.

Como la base estructural del paradigma es la recursión, en estos párrafos se presenta lo que se conoce como *el árbol de recursión* de un algoritmo.

Un árbol de recursión es una manera abstracta de representar las dependencias recursivas de un algoritmo. Se dice que es un árbol debido a que puede ser representado como una gráfica conexa, acíclica (que no presenta ciclos) y no dirigida, lo cual cumple con la definición de un *árbol* dentro de la Teoría de Gráficas.

El problema original a resolver es considerado la *raíz* del árbol y a partir de él se comienza a crear el árbol. Por cada llamada recursiva necesaria para resolver el problema actual se agrega un nuevo nodo como hijo del nodo actual. De este modo, el árbol va creciendo tanto hacía abajo como a los lados. El árbol se completa cuando se llega a problemas que son considerados casos base pues estos ya no cuentan con llamadas recursivas por lo que de ellos ya no se deriva ningún nodo en el árbol. Las hojas del árbol son entonces los casos base del problema original mientras que los nodos internos son problemas que se resuelven de manera recursiva.

Hay que notar que este árbol de recursión tiene un nodo distinguido que es considerado la raíz. Así mismo, las hojas de este árbol tienen un significado especial y el árbol se debe interpretar desde la raíz hacía las hojas considerando, en un sentido figurado, la noción de dirección. La Figura 3.1 muestra la representación gráfica de un árbol de recursión.

Se observa que con esta representación se obtienen las dependencias recursivas de los subproblemas derivados de uno más complejo. Por ejemplo, si un nodo P es padre de un nodo Q, entonces Q es un subproblema generado a partir de P; esto implica que para poder resolver P se necesita primero resolver Q. Así, dos nodos hermanos en el árbol representan dos subproblemas generados a partir del problema original y no tienen relación entre sí, mas que su origen, por lo que en principio podrían resolverse en cualquier orden sin afectar el resultado (esto se comprobará a lo largo del trabajo).

3.1.2. División en subproblemas.

Al utilizar este pardigma se debe considerar dividir al problema original en dos o más subproblemas que sean más sencillos de resolver. Esto por supuesto induce de manera natural a la recursión. Sin embargo, antes de poder crear los subproblemas se debe determinar la manera en que se va a dividir el problema original.

El determinar la manera de dividir el problema en subproblemas a veces no es simple y se tienen que hacer cálculos para poder lograrlo. Como se verá más adelante, esta manera depende del problema a resolver y depende de las suposiciones que se hagan acerca del conjunto de trabajo del problema a tratar.

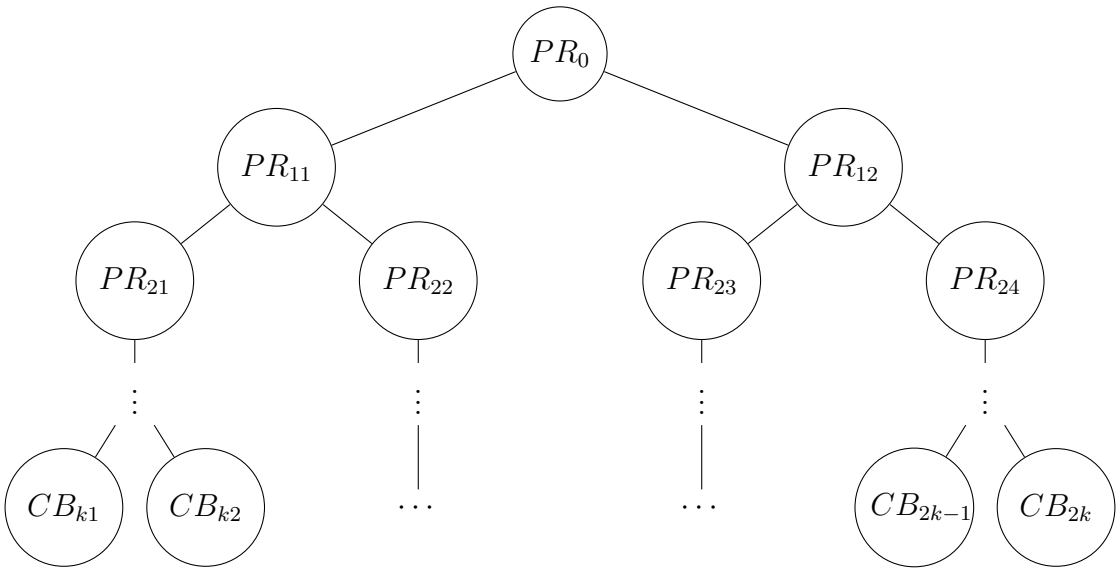

Figura 3.1: Representación gráfica de un árbol de recursión. PR significa que para resolver ese problema se deben hacer llamadas recursivas y CB significa que es un caso base del problema por lo que ya no son necesarias las llamadas recursivas.

Esta etapa de división es bajar en los niveles del árbol de recursión del problema y el costo que tiene bajar[1] un nivel del árbol es el costo de calcular los subproblemas. Se entiende por costo el número de operaciones necesarias para llevar a cabo la tarea.

3.1.3. Resolver los subproblemas.

La manera más sencilla de resolver los subproblemas es mediante recursión. Se invoca recursivamente el algoritmo que se está creando sobre cada uno de los subproblemas que se han definido en el paso anterior.

Siempre que se trabaja con recursión es necesario definir uno o más casos base. Como ya se mencionó anteriormente, para este tipo de algoritmos, los casos base son aquellos subproblemas pequeños (sencillos) para los cuales la solución es simple o, sino lo es, encontrarla requiere de un número constante de operaciones.

3.1.4. Combinar las soluciones.

Gracias a estos casos base se pueden resolver problemas más grandes y complicados de manera recursiva. Sin embargo, la gran mayoría de las veces se necesita aplicar operaciones adicionales a las soluciones de los subproblemas para poder construir la solución del problema original.

Estas operaciones por lo general no son simples y su uso de recursos depende del tamaño de la entrada. Por lo que la esencia de estos algoritmos está en combinar las soluciones de los

[1]Bajar se refiere a considerar ahora como problema original a cada uno de los hijos del nodo actual.

subproblemas. Esta es la *operación clave* en los algoritmos *divide y vencerás*.

Combinar las soluciones de los subproblemas es subir[2] un nivel en el árbol de recursión del problema y el costo de hacerlo es el costo necesario para subir de un nivel al siguiente en el árbol.

3.2. Usos y ejemplos.

En esta sección se presentan varios ejemplos de algoritmos secuenciales conocidos diseñados con este paradigma. Por medio de la descripción de dichos algoritmos se puede observar la gran variedad de campos de estudio en los que este paradigma se ha aplicado.

3.2.1. Algoritmo de ordenación *quicksort*.

Este es un famoso algoritmo de ordenación desarrollado por Sir Anthony Hoare en el que se pretende ordenar una secuencia de elementos totalmente comparables entre sí. La descripción se encuentra en el Algoritmo 3.1.

Algoritmo 3.1 Quicksort.

Entrada A es una secuencia de elementos totalmente comparables entre sí.
Salida Una secuencia ordenada con exactamente los mismos elementos de A.

function QUICKSORT(A)
  if $|A| \leq 1$ **then**
    return A.
  end if
  Escoger un elemento de A llamado *pivote*.
  Construir la secuencia A_{men} que consta de todos los elementos de A menores o iguales que *pivote*.
  Construir la secuencia A_{may} que consta de todos los elementos de A mayores que *pivote*.
  $Res_{men} \leftarrow quicksort(A_{men})$.
  $Res_{may} \leftarrow quicksort(A_{may})$.
  Concatenar Res_{men} seguido de Res_{may} en un nuevo arreglo A'.
  return A'.
end function

En el algoritmo la construcción de las secuencias A_{men} y A_{may} es la parte de dividir. Se observa que en este algoritmo se crean dos subproblemas a resolver. La recursión es la etapa de vencer. Se observa que el caso base es cuando se quiere ordenar una secuencia de uno o menos elementos y es claro ver que dicha secuencia ya está ordenada, por lo que la solución del caso base ya está dada. Finalmente, la parte de concatenar las soluciones es el tercer elemento del paradigma: la fase de combinar. En este ejemplo con solo concatenar las soluciones de los subproblemas se obtiene la solución del problema original.

[2]Subir se refiere a volver a considerar como problema original al padre de todos los nodos con los que se está trabajando.

Este algoritmo de ordenación es uno de los más usados en la práctica por ser rápido y fácil de implementar en la mayoría de los lenguajes, aunque su complejidad en el peor caso no es la óptima ya que se requiere tiempo $O(n^2)$. Sin embargo, el tiempo esperado de ejecución utilizando una sencilla modificación al algoritmo presentado aquí es $\Theta(n \log\ n)$, [9].

3.2.2. Multiplicación de números enteros.

Este algoritmo multiplicará dos números en base 2 que tienen demasiadas cifras como para caber en un registro de la máquina. La idea central de este algoritmo es representar los números a multiplicar como suma de dos números con menos cifras y multiplicarlos recursivamente obteniendo la solución tras combinar las soluciones. Esta técnica fue creada por Anatoly Karatsuba y se encuentra en el Algoritmo 3.2.

Algoritmo 3.2 Multiplicación de Números Enteros.

Entrada x e y son los números a multiplicar en base 2.
Salida La multiplicación de x por y.
  function MULTIPLICARECURSIVO(x,y).
    Reescribe $x = x_1 \cdot 2^{n/2} + x_0$.
    Reescribe $y = y_1 \cdot 2^{n/2} + y_0$.
    Calcula $x_1 + x_0$ y $y_1 + y_0$.
    $p \leftarrow multiplicaRecursivo(x_1 + x_0, y_1 + y_0)$.
    $x_1y_1 \leftarrow multiplicaRecursivo(x_1, y_1)$.
    $x_0y_0 \leftarrow multiplicaRecursivo(x_0, y_0)$.
    return $x_1y_1 \cdot 2^n + (p - x_1y_1 - x_0y_0) \cdot 2^{n/2} + x_0y_0$.
  end function

En este algoritmo la etapa de división se realiza al expresar los números x e y como suma de dos números con menos dígitos. En este caso x_1 e y_1 representan a la mitad de bits más significativa de los números originales. Una vez calculados estos nuevos números se procede a invocar de manera recursiva a la misma función sobre números de menos dígitos. Es importante observar que en este algoritmo como tal no hay un caso base, el caso base se obtiene cuando los números a multiplicar caben en un registro de la computadora[3]. La etapa de combinación se obtiene haciendo operaciones aritméticas sobre los resultados obtenidos en la etapa recursiva.

Este algoritmo tiene una complejidad de $O(n^{\log_2 3}) \approx O(n^{1.584963})$ que es estrictamente menor que la complejidad cuadrática del algoritmo ingenuo, [14].

3.2.3. Multiplicación de matrices.

En esta sección se muestra un algoritmo para multiplicar matrices. Este método fue inventado por Volker Strassen y lleva su nombre. El algoritmo multiplica dos matrices cuadradas utili-

[3]Los registros de las computadoras actuales tienen 64 bits de longitud, por lo que cualquier número con a lo más 64 dígitos en representación binaria cabe en dichos registros.

zando una técnica ingeniosa y por supuesto, el paradigma *divide y vencerás*. El pseudocódigo se presenta en el Algoritmo 3.3, [9].

Algoritmo 3.3 Algoritmo de Strassen para multiplicación de matrices.

Entrada A y B son dos matrices de $n \times n$, con n una potencia de dos.
Salida La matriz $C = A \cdot B$.
Sea C una nueva matriz de $n \times n$.
if $n == 1$ **then**
$C_{11} = A_{11} \cdot B_{11}$.
end if
Dividir A de la siguiente manera $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$.
Dividir B de la siguiente manera $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$.
Dividir C de la siguiente manera $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$.
Crear las matrices $S_1 \dots S_{10}$ de la siguiente manera:
$S_1 = B_{12} - B_{22}$, $S_2 = A_{11} + A_{12}$, $S_3 = A_{21} + A_{22}$, $S_4 = B_{21} - B_{11}$, $S_5 = A_{11} + A_{22}$, $S_6 = B_{11} + B_{22}$, $S_7 = A_{12} - A_{22}$, $S_8 = B_{21} + B_{22}$, $S_9 = A_{11} - A_{21}$, $S_{10} = B_{11} + B_{12}$.
Utilizando estas submatrices calcular recursivamente:
$P_1 = A_{11} \cdot S_1$, $P_2 = S_2 \cdot B_{22}$, $P_3 = S_3 \cdot B_{11}$, $P_4 = A_{22} \cdot S_4$, $P_5 = S_5 \cdot S_6$, $P_6 = S_7 \cdot S_8$, $P_7 = S_9 \cdot S_{10}$.
$C_{11} = P_5 + P_4 - P_2 + P_6$.
$C_{12} = P_1 + P_2$.
$C_{21} = P_3 + P_4$.
$C_{22} = P_5 + P_1 - P_3 - P_7$.

En el algoritmo se observa que cuando la dimensión es 1, se aplica el caso base. Pero antes de poder invocar la rutina recursiva se necesitan hacer una gran cantidad de cálculos adicionales. Esto es prueba que hay ocasiones en donde la etapa de dividir no es sencilla y puede requerir muchas operaciones. A pesar de lo anterior y de la gran cantidad de espacio requerido para almacenar las matrices auxiliares, este algoritmo tiene una complejidad de $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.807355})$ que mejora asintóticamente al usual que tiene complejidad temporal $O(n^3)$, [9].

3.3. Ventajas y desventajas.

Las principales ventajas del paradigma *divide y vencerás* radican en simplificar un proceso complejo fragmentándolo en subprocesos más simples, lo que es apoyado en la recursión. Gracias a la recursión la mayoría de estos algoritmos son bastante fáciles de explicar y sencillos de entender, pues una vez que se entiende el concepto abstracto de la recursión, para entender el algoritmo en su totalidad, basta con entender la etapa de división en subproblemas y la etapa de combinar las soluciones. En otras palabras, si uno está familiarizado con el concepto de recursión, la etapa de *vencer* se entiende de manera inmediata.

Cabe señalar que la mayoría de estos algoritmos mejoran a un algoritmo natural o simple para resolver un problema. Si se analizan los ejemplos de la sección anterior, se concluye que hay otras maneras de resolverlos y que, de hecho, el algoritmo presentado aquí dista mucho de ser la manera intuitiva de hacerlo. Sin embargo, al utilizar este paradigma se obtiene un algoritmo que es más eficiente, aunque, a veces, más complicado de entender e implementar.

Al hacer uso fundamental de la recursión, estos algoritmos sufren de las mismas desventajas que otros algoritmos recursivos. Por ejemplo, el uso de la pila del sistema para almacenar llamadas recursivas, fundamentar e implementar correctamente los *casos base* o incluso concebir la manera recursiva de resolver un problema dado.

Al diseñar estos algoritmos también se deben tener en cuenta aspectos como la eficiencia de las etapas de *división* y, si es necesario, hacer cálculos para dividir el problema como se verá más adelante. En particular, se debe siempre tener en mente el costo total del algoritmo ingenuo o simple pues pudiera darse el caso que solo el costo y esfuerzo para dividir el problema sea equiparable con el costo y esfuerzo de resolver el problema en su totalidad con dichos algoritmos.

Esto mismo aplica para la etapa de *combinar* los resultados. Hay ocasiones en las cuales esta etapa solo utiliza operaciones sencillas sobre los resultados de los subproblemas; sin embargo, hay otras en las que se necesita hacer una gran cantidad de cálculos y operaciones adicionales para poder obtener la solución total y las soluciones de los subproblemas solo sirven para restringir la posible solución total.

3.4. Divide y vencerás: de lo recursivo a lo iterativo.

Hasta ahora se ha presentado el paradigma *divide y vencerás* de manera recursiva, es decir, con un enfoque de *arriba hacia abajo* (en inglés *Top-Down*). Sin embargo, para los casos de estudio los algoritmos que se presentan están basados en una aproximación de *abajo hacia arriba* (en inglés *Bottom-Up*), así que en los siguientes párrafos se describe esta otra aproximación de *divide y vencerás*.

Hay ocasiones en que no se puede (o no se quiere) usar recursión, ya sea porque la plataforma no lo permite o porque es muy costosa. Sin embargo, se quiere usar un algoritmo *divide y vencerás*. Entonces surge la pregunta de si acaso es posible esto, pues en principio podría parecer que la recursión es inherente al paradigma ya que ¿cómo podría existir la etapa de *vencer* sin utilizar recursión?

Sin embargo, al analizar la estructura de estos algoritmos se concluye que la idea fundamental es dividir el problema en problemas más sencillos y luego combinar las soluciones en una más compleja, pero ¿qué pasa si desde un inicio se tienen las soluciones a los problemas más sencillos? Si este fuera el caso, entonces ya no es necesario dividir el problema y solo se requiere unir las soluciones. Hay que recordar también que para resolver los casos base solo se necesitan hacer operaciones sencillas.

Con esto en mente ya se puede tener otra aproximación para resolver el problema original ya

que primero se pueden resolver los casos base y luego combinar estas soluciones de la manera usual para crear una solución más compleja y de manera *iterativa* ir combinando soluciones sencillas ya creadas para formar una solución más compleja. Esto es el proceso de subir en el árbol de recursión descrito anteriormente; sin embargo, esta nueva aproximación es puramente iterativa pues si se conoce el tamaño del problema original, de acuerdo a la estructura del árbol se puede calcular el número máximo de iteraciones (niveles del árbol de recursión) por lo que se puede proseguir de esta manera hasta resolver el problema original.

Esta estrategia no es propia de este paradigma, sino que se puede hacer con cualquier algoritmo recursivo. Aquí la se describe de manera simple para algoritmos *divide y vencerás* pero en cada caso de estudio se hace uso explícito de esta nueva aproximación para resolver el problema dado.

Así como la aproximación recursiva combina las soluciones tras haber dividido el problema, este nuevo enfoque requiere lo mismo. Antes de poder comenzar a iterar para resolver los subproblemas se debe de hacer una etapa de división global, tal y como se verá en los casos de estudio. Es decir, primero se debe especificar claramente cómo dividir el problema original en casos base y cómo usar los conjuntos de trabajo correspondientes en cada etapa de iteración. Cuando la etapa de división es sencilla, esta parte también lo es y no se ve reflejada en el algoritmo; sin embargo, si la etapa de división no es simple, se debe lidiar con este problema antes de poder resolver los subproblemas.

Debido a que la arquitectura de las GPU no está diseñada para hacer óptima la recursión se utiliza la aproximación iterativa para programar los casos de estudio en las GPU.

Capítulo 4

Conceptos de paralelización.

En este capítulo se presentan algunos conceptos teóricos y prácticos acerca de la paralelización de programas o tareas. Los conceptos aquí presentados ayudarán a paralelizar de mejor manera los algoritmos expuestos en los casos de estudio.

En la primera sección se discute una expresión matemática que describe una limitación inherente a la mejora que se presenta en un programa paralelo comparado con una versión secuencial. En la segunda sección se expone una limitante lógica a la paralelización. Esta limitante determina el orden en el que se tienen que procesar las subtareas para obtener el resultado de una tarea más compleja. En la tercera sección se presentan los conceptos de sincronización que permiten procesar las tareas en el orden dictado por las restricciones de la segunda sección. Estos conceptos también evitan la corrupción de datos en los programas paralelos.

En la última sección se precisa un concepto aplicable a los algoritmos aquí estudiados. Debido a la naturaleza recursiva de dichos algoritmos y a la limitante de arquitectura en CUDA, estos se deben implementar de forma iterativa y no recursiva. Esto conlleva a saber la manera exacta de dividir el problema para poder paralelizarlo. En esa sección se establecen las maneras de conocer dichas divisiones: en tiempo de compilación, o en tiempo de ejecución. También se discuten las ventajas y desventajas de estas divisiones.

4.1. Ley de Amdahl.

En 1967 el arquitecto de computadoras Gene Amdahl presentó una relación matemática que determina la máxima mejora en tiempo que puede tener un sistema cuando solo una parte de dicho sistema es mejorada. Esta relación es llamada, en su honor, como *la ley de Amdahl*.

Esta ley determina la mejora en rendimiento que tendrá un sistema al ser paralelizado, por lo que cuantifica matemáticamente la mejora en el desempeño de los programas al ser paralelizados. La ley de Amdahl dice lo siguiente:

Sea n, $n \in \mathbb{N}$, el número de hilos en ejecución, sea S, $S \in [0,1]$, el porcentaje total da la parte del código que es secuencial. Sea $T(1)$ el tiempo estimado de la ejecución secuencial, entonces el tiempo $T(n)$ es el tiempo estimado de ejecución de un algoritmo ejecutado con n hilos de ejecución y se calcula de la siguiente manera:

$$T(n) = T(1)\big(S + \tfrac{1-S}{n}\big)$$

Es decir, el tiempo que toma ejecutar un programa paralelo es el tiempo que toma realizar la parte secuencial más el tiempo que toma realizar la porción de código paralela dividida entre el número de hilos utilizados.

La *mejora* (en inglés speedup) $M(n)$ que puede alcanzar un programa al ser paralizado con n hilos es la proporción entre el tiempo de ejecución secuencial entre el tiempo de ejecución con n hilos y se define como:

$$M(n) = \frac{T(1)}{T(n)} = \frac{1}{S+\frac{1}{n}(1-S)}$$

Por ejemplo: si se puede paralelizar el 40 % de un programa secuencial, entonces $S = 0.6$, mientras que el código paralelo se ejecuta con 8 hilos, que es algo factible en una computadora personal de hoy día, entonces la mejora en rendimiento obtenida es:

$$M(8) = \frac{1}{0.6+\frac{1}{8}(0.4)} = \frac{1}{0.6+0.05} \approx 1.5338$$

Entonces este programa, en teoría, sería 1.53 veces más rápido que el programa secuencial. Hay que notar que cuando el número de hilos tiende a infinito, la mejora tiende a $\frac{1}{S}$. Por lo que si S es 0.1 (suponiendo que el 90 % del programa pudo paralelizarse), entonces la mejora es de 10, por lo que el programa paralelo será 10 veces más rápido que el secuencial. Esto sin importar qué tantos hilos se usen.

Hay problemas que poseen un alto grado de paralelismo y se ven beneficiados cuando se incrementa el número de hilos. Sin embargo, llega un momento en que si se incrementa más el número de hilos, el desempeño decae, ya que el costo administrativo de todos los hilos es mayor que la mejora obtenida al paralelizar el código.

4.2. Dependencia de tareas.

Hay muchos problemas en los que la obtención de un resultado complejo depende de resultados más simples, por lo que los resultados simples deben de obtenerse primero y mientras dichos resultados sean aún desconocidos, entonces el resultado complejo no puede obtenerse.

Un ejemplo de lo anterior son justo los tipos de algoritmos tratados en este trabajo. Los algoritmos *divide y vencerás* tienen la cualidad de componer resultados sencillos en unos más complejos, por lo que al paralelizarlos se debe tener en cuenta que un resultado complejo se puede calcular solo hasta que los resultados más sencillos estén ya calculados. Esto define una *dependencia de tareas* en el aspecto lógico de la paralelización.

Recordando el árbol de recursión presentado en el capítulo anterior, se pueden definir dos tipos de tareas. Las tareas *independientes* corresponden a las hojas del árbol, es decir, a los casos base del problema, pues dichas tareas no dependen del resultado de una subtarea para poder ser calculadas. Las otras tareas corresponden al resto de los nodos del árbol, los nodos internos son tareas *dependientes*, pues requieren que las subtareas, que de ellos emanan, se resuelvan antes para poder calcular su propio resultado.

4.2.1. Gráfica de dependencias.

Álcántara define el concepto de gráfica de dependencias para problemas de programación dinámica[1], [5]. Esta noción se puede extender a los algoritmos *divide y vencerás* pues a partir del árbol de recursión se puede definir una *gráfica acíclica dirigida* (más conocidas por sus siglas en inglés: *DAG* (*Directed Acyclic Graph*)) de la siguiente manera:

Definición 4.2.1 (Digráfica de dependencias de un problema *divide y vencerás*)**.** *Sea* $T = (V, A)$ *el árbol de recursión de un problema divide y vencerás. Definimos a* D*, la digráfica de dependencias del problema como:* $V(D) = V(T)$ *y* $A(D) = \{(x, y),\ x, y \in V(D) \mid x$ `es padre de` y `en` $T\}$

Una vez definida la digráfica de dependencias se pueden utilizar los conceptos definidos por Alcántara y aplicarlos a esta digráfica de dependencias ya que dichos conceptos se basan en la teoría de gráficas. Uno de esos conceptos llamado *etapa* habla acerca del conjunto de nodos que están a la misma distancia desde la raíz del árbol. Como ya se mencionó, para transformar un algoritmo *divide y vencerás* en forma iterativa se deben resolver primero los casos base y luego ir combinando las soluciones para subir en el árbol de recursión. Esta noción se traspasa a la digráfica de dependencias pues la *etapa* compuesta por los casos base se debe resolver primero y una vez resueltos, estos nodos se pueden eliminar de la digráfica para generar una nueva. Se puede seguir con este proceso sobre las nuevas hojas hasta haber resuelto todos los nodos quedando solo la raíz y en este momento ya se puede generar la solución al problema original pues ya se tienen resueltos todos los subproblemas.

También se observa que todas las hojas pueden resolverse independientemente, de manera paralela. Específicamente, si P es hermano de Q, entonces P no depende de Q por lo que el orden en el que se resuelvan no importa, así que P y Q pueden ser resueltos de forma paralela.

Con esto se ha establecido un orden lógico en el que se pueden resolver en paralelo los subproblemas de un problema *divide y vencerás*. Dos subproblemas se pueden paralelizar si y solo si uno no es *ancestro* del otro en el árbol de recursión. Es decir, si no están en la misma trayectoria desde la raíz del árbol.

Cabe mencionar que la biblioteca de Intel *Intel Threading Building Blocks* [4], que es una biblioteca escrita en C++ para poder realizar algoritmos y tareas paralelas, implementa un objeto *graph* que simula la digráfica de dependencias de una tarea paralela. Por medio de instrucciones se establecen las dependencias entre objetos y la digráfica se ejecuta tantas veces

[1] La programación dinámica es un paradigma para la resolución de problemas tal y como lo es *divide y vencerás*.

como etapas tenga para poder llevar a cabo los cálculos. Esta biblioteca está enfocada en definir tareas y abstraer el paralelismo del lado del programador, por lo que esta aproximación es muy fácil de utilizar; sin embargo, crear la digráfica de dependencias explícitamente es útil y eficiente solo cuando la digráfica tiene pocas aristas, pues el costo administrativo para crear y mantener la digráfica es proporcional a su tamaño (la suma del número de vértices más el número de aristas).

4.3. Sincronización de tareas.

Cuando se divide una tarea para ser realizada de manera paralela, surge inherentemente la necesidad de comunicación entre los hilos involucrados. Esto tiene que ver no solo con el orden dado por la gráfica de dependencias de la tarea sino también por la necesidad de manipular recursos compartidos. Si se imagina una curva muy amplia en una vía de tren donde no se alcanzan a ver sus extremos, y por esta curva tienen que pasar trenes en ambos sentidos, dado que solo hay una vía, no puede haber dos trenes en la curva al mismo tiempo pues podría ocurrir una colisión. Es entonces cuando los trenes deben tener un protocolo para transitar por la curva de manera que se asegure que solo habrá un tren a la vez. Se dice que esta sección de vía es una *sección crítica* y los procesos involucrados (en este caso trenes) deben sincronizarse para que solo un proceso utilice la sección crítica a la vez. A este problema se le conoce como *exclusión mutua* y es un ejemplo común para presentar la noción de recurso compartido y corrupción de ese recurso.

En procesos paralelos es necesario idear mecanismos de comunicación y sincronización entre procesos para manejar los recursos compartidos y para satisfacer las dependencias de tareas del problema. Como ejemplo de esto, supóngase un taller de carpintería en el que los carpinteros quieren construir un comedor. Uno hace la tabla de la mesa y otros las patas. Si antes se ponen de acuerdo en el diseño, estas tareas son paralelas; sin embargo, para hacer la mesa completa deben tener la tabla y todas las patas para poder fijarlas a la tabla, por lo que la tarea de "pegado" depende de las tareas de creación de las piezas. Así que aunque haya un carpintero listo para fijar la mesa, si las partes aún no están listas, este carpintero debe esperar (tal vez realice otro trabajo mientras). En este ejemplo el problema es la dependencia de tareas y la espera inherente para poder llevar a cabo ciertas tareas. En este caso, por ejemplo, es inviable asignar a un carpintero para ensamblar la mesa desde el inicio pues se podría quedar esperando por las partes mucho tiempo y desperdiciar ese tiempo en el que podría realizar otras actividades.

Es por estas limitaciones inherentes al cómputo paralelo, no presentes en el cómputo secuencial, que se han ideado mecanismos de sincronización entre procesos a lo largo de la historia y aunque estos mecanismos varían entre arquitecturas paralelas, siempre están presentes en las mismas. A continuación se describen los mecanismos de sincronización provistos por la arquitectura CUDA y que se utilizarán en los casos de estudio.

4.3.1. Métodos de sincronización en CUDA.

Aquí se describe el uso y la noción teórica de los mecanismos de sincronización en CUDA; sin embargo, no se discute la implementación ya sea en hardware o software de los mismos.

Sincronización de barrera.

La sincronización de barrera funciona de la siguiente manera: se quiere definir un *punto de encuentro* de los procesos involucrados en una tarea, de tal forma que ningún hilo continúe su ejecución hasta que todos los demás hilos hayan llegado al punto de encuentro. Este punto de encuentro actúa como una barrera que sincroniza a los procesos pues permite asegurar que todas las instrucciones anteriores al punto han sido ejecutadas completamente antes de que empiece a ejecutarse una instrucción después de la barrera por cualquier hilo.

En CUDA se provee una instrucción nativa para usar sincronización de barrera entre hilos del mismo bloque. Dicha instrucción es `__syncthreads()`. Esta es una sincronización de barrera para hilos del mismo bloque y asegura que ningún hilo del bloque empezará a ejecutar instrucciones posteriores a esta hasta que todos los hilos hayan ejecutado el `__syncthreads()`.

Cabe señalar que CUDA no provee un mecanismo de sincronización de barrera a nivel global (entre bloques) por lo que se discutirá más adelante una manera de obtener dicha sincronización.

Operaciones atómicas.

Una *operación atómica* es una instrucción que no puede ser interrumpida ya sea por hardware o por software. Por interrumpir se entiende que una vez que la instrucción comenzó a ejecutarse, esta terminará su semántica sin que otro hilo modifique los registros o la memoria que estén en uso por el hilo que ejecuta la instrucción. Un hilo que ejecute una operación atómica tiene la certeza de que su operación habrá sido completada sin que otro hilo modifique los datos que se están usando en ella. Las operaciones atómicas se pueden pensar como indivisibles y siempre garantizan que o son ejecutadas por completo o no son ejecutadas en sí.

Las operaciones atómicas permiten un mecanismo sencillo de sincronización y están presentes en muchas arquitecturas paralelas e incluso en arquitecturas secuenciales. Herlihy las presenta de manera más formal y teórica, [11].

Hay que tener en cuenta que las instrucciones atómicas son fáciles de entender y utilizar pero son muy costosas, pues en primera, las que se implementan en hardware pueden llegar a bloquear toda la memoria e invalidar líneas de todos los cachés para asegurar su “atomicidad” y consistencia. Esto, sin contar que requieren hardware especial que las implemente. Otra razón por la que son costosas es que, aunque a nivel de biblioteca se tenga un amplio catálogo de dichas instrucciones, hay arquitecturas que no implementan todas ellas a nivel de hardware, por lo que la implementación en la biblioteca es por medio de otras más simples que sí están presentes en el hardware.

En CUDA se cuentan, entre otras, con las siguientes instrucciones atómicas:

- `int atomicAdd(int* address, int val)`: suma un valor al valor contenido en una dirección de manera atómica.
- `int atomicExch(int* address, int val)`: intercambia dos valores en una dirección.
- `int atomicMax(int* address, int val)`: calcula el máximo entre un valor y el contenido de la dirección y almacena el máximo en la dirección.
- `int atomicCAS(int* address, int compare, int val)`: la instrucción *compare and swap* compara el valor de la dirección con un argumento y en caso de que sean el mismo se almacena el valor `val` en la dirección. En caso de que la comparación fallé (es decir, los valores sean distintos) el valor de la dirección se deja sin cambios.

En la guía de programación de CUDA se encuentran detalladas todas las operaciones atómicas implementadas, [3].

Sincronización global.

Teóricamente las operaciones atómicas pueden usarse para sincronizar a todos los bloques de manera global. Por ejemplo, hacer que un representante de cada bloque aumente de manera atómica un contador para contar los bloques que han terminado su labor y hacer que todos los hilos esperen a que el contador sea un valor dado. Sin embargo, en la especificación de CUDA no se garantiza que el calendarizador de bloques intercambie un bloque por otro en algún momento, por lo que podría ocurrir que un bloque se quedara ejecutando indefinidamente esperando a los demás sin oportunidad de que los otros actualicen esta variable por lo que ocurriría un *interbloqueo* (en inglés *deadlock*). En caso de los algoritmos *divide y vencerás* hay otra manera de sincronizar globalmente la tareas.

Estos algoritmos se ejecutarán por etapas (las cuales están dadas por el árbol de recursión y la gráfica de dependencias). En cada etapa se debe asegurar una sincronización para que las dependencias de tareas se satisfagan. Dado que usualmente se tienen varios bloques en ejecución se debe hallar un mecanismo de barrera de sincronización de manera global. Este mecanismo no es provisto por la arquitectura como un mecanismo de sincronización sino que está implícito en el diseño de la misma.

En cada etapa se lanza un *kernel* para procesarla. Sin embargo, se debe esperar a que el *kernel* que procesa la etapa i termine por completo para que se lance el *kernel* que procesa la etapa $i+1$. CUDA C permite lanzar varios *kernels* uno tras de otro de manera secuencial; es decir, si se lanza un *kernel* k en un dispositivo y luego en C se lanza otro *kernel* l en el mismo dispositivo, el *kernel* l iniciará su ejecución hasta que el *kernel* k haya terminado su ejecución. Este diseño es justo el deseado pues la "secuenciación" de los lanzamientos permite sincronizar las dependencias de tareas de manera global.

Este tipo de sincronización, junto con la sincronización de barrera entre hilos del mismo bloque son los mecanismos más utilizados en los casos de estudio.

4.4. Paralelismo en algoritmos *divide y vencerás*.

En esta sección se discutirá acerca de cómo calcular los subproblemas de una tarea para poder paralelizarlos, en general se discutirá si se pueden calcular en tiempo de compilación o no.

4.4.1. Paralelismo precalculable y no precalculable.

Como ya se ha mencionado, en los problemas *divide y vencerás*, la etapa de dividir consiste en partir al problema original en varios subproblemas estructuralmente más sencillos. En particular cuando se trata de manera iterativa estos problemas esta división debe ya estar hecha, pues las etapas iterativas equivalen a las etapas de vencer y combinar.

Sin embargo, ¿cómo se divide al problema en sus subproblemas antes de poder iterar sobre ellos para poder calcular la solución total? La idea es usar la gráfica de dependencias y fijarse en los casos base, después en sus padres y así hasta llegar al nodo raíz, que es el problema original. Con la noción de etapa se puede paralelizar y dividir la solución del problema en tantas etapas como tenga la gráfica de dependencias, siendo la etapa de los casos base la primera y la etapa de solución del problema original la última.

Sin embargo, para lograr lo anterior se debe tener la gráfica de dependencias o en su defecto, una manera de calcularla, para ciertos problemas esto es relativamente sencillo: por ejemplo, si el problema solo requiere dividir a la mitad la entrada y hacer recursión sobre cada parte. Este tipo de divisiones son muy sencillas y pueden calcularse en tiempo de compilación; es decir, dejar indicado en una función en código, cómo se dividirá la entrada y en ese momento se sabe, por ejemplo, que el tamaño de los ejemplares más pequeños es la mitad del tamaño del original.

Si se da el caso anterior al momento de hacer una implementación iterativa se puede pensar que la entrada original ya está dividida y pasar directo a la etapa de solución. Sin embargo, hay problemas como *merge*, presentado en el caso de estudio, donde para dividirlo en subproblemas se debe calcular a todos los elementos del problema original que cumplan cierta propiedad. La división es entonces:
Dado S, la entrada original, y P una propiedad de los elementos de S, se definen
$S_1 = \{x \in S | P(x)\},\ S_2 = \{x \in S | \neg P(x)\}$

Con problemas como este, la etapa de dividir no es sencilla. Por ejemplo, en el Algoritmo 3.1 (Quicksort) del Capítulo 4 se necesita procesar la entrada para poder saber de qué tamaño serán los subproblemas, por lo que no se puede decir en tiempo de compilación siquiera si habrá dos subproblemas (en particular si el elemento elegido como *pivote* resulta ser el más grande de la lista de entrada). Ante este tipo de problemas se debe efectuar una etapa de división antes de poder pasar a procesar los subproblemas para combinarlos y obtener la solución general.

A pesar de que en estos problemas también existe paralelismo, este paralelismo se presenta hasta que se han calculado los conjuntos de elementos que satisfagan los criterios de separación (en el ejemplo el criterio es una propiedad y separa en dos al conjunto original, pero

puede haber problemas donde los criterios creen más subconjuntos). Se dice que este paralelismo es *no precalculable* pues no puede ser calculado en tiempo de compilación a partir del conjunto de entrada ya que se necesitan hacer cálculos en tiempo de ejecución para conocer los subproblemas paralelos. Por otro lado, el paralelismo *precalculable* es aquel en donde se conocen exactamente los ejemplares paralelos en tiempo de compilación (problemas cuya etapa de división es simple como partir en k partes la entrada, por ejemplo).

El algoritmo de *merge* en el caso de estudio exhibe paralelismo no precalculable y se discutirá más a fondo la estrategia de paralelización en el capítulo correspondiente.

Capítulo 5

Caso de estudio: transformada rápida de Fourier.

En este capítulo se presenta el primer algoritmo realizado en CUDA C. El algoritmo que calcula la transformada rápida de Fourier (en inglés *Fast Fourier Transform*) exhibe un buen paralelismo para ser implementado en una arquitectura masivamente paralela.

En las siguientes secciones se describe el algoritmo secuencial y las aplicaciones que tiene. Después se discute la estrategia de paralelización, es decir, los elementos y propiedades que permiten paralelizar el algoritmo de manera efectiva. Finalmente se muestra el código en CUDA C y los resultados y conclusiones obtenidos tras haber implementado el algoritmo.

5.1. Motivo de estudio.

La transformada de Fourier, llamada así en honor al matemático Joseph Fourier, es una transformación matemática que expresa una función en el dominio de la frecuencia a partir del dominio del tiempo. Esta transformada tiene su principal aplicación en computación dentro del área de procesamiento digital de señales.

Dado que la computadora trabaja sobre dominios discretos, *la transformada discreta de Fourier* (en inglés *Discrete Fourier Transform*) es una transformación de Fourier que permite expresar un colección de muestras espaciadas uniformemente en una lista de coeficientes senoidales complejos. De nuevo, sus principales aplicaciones están en el procesamiento digital de señales.

La *transformada rápida de Fourier* es un algoritmo que implementa la noción de la transformada discreta y su inversa. En este trabajo se utiliza la transformada rápida de Fourier en una aplicación que no solo compete al área de procesamiento digital de señales: la multiplicación de polinomios.

5.2. Planteamiento del problema.

La noción de polinomio está presente en varias áreas de las matemáticas y de la computación. En álgebra, por ejemplo, se estudian las propiedades de los polinomios y sus operaciones como espacios algebraicos (campos, grupos, anillos, espacios vectoriales), en el análisis se estudian las propiedades de las funciones que definen los polinomios.

En computación la noción de polinomio se utiliza en el área de la criptografía, del análisis numérico, del procesamiento digital de señales, entre otros.

Dado que los polinomios son muy conocidos y utilizados, estos objetos matemáticos deben de ser implementados de manera eficiente en la computadora, pero no solo se necesita una representación eficiente de ellos, sino también algoritmos eficientes que permitan modificarlos y operar con ellos. Entre las operaciones más usadas se encuentran, la suma, la evaluación y la multiplicación.

Para representar polinomios hay dos maneras básicas y bastante sencillas. La primera es representar a un polinomio de grado $n-1$ por sus n coeficientes. La segunda es representarlo, escogiendo n números distintos y evaluar el polinomio en ellos, obteniendo n parejas de la forma $(x, P(x))$ donde x es el número y $P(x)$ la evaluación del polinomio a representar en x. Si para todos los polinomios de grado $n-1$ con coeficientes en un campo C se eligen los mismos n números para realizar las evaluaciones, entonces se pueden representar de una única manera a todos los polinomios en el campo. Para multiplicar polinomios usando la transformada rápida de Fourier se elige esta última manera de representarlos.

Con la representación por lista de coeficientes, la suma se puede implementar con complejidad $\Theta(n)$ donde n es el grado más grande de los dos polinomios a sumar. Por otro lado la evaluación de un polinomio P en un valor x se puede obtener también con complejidad $\Theta(n)$ utilizando la regla de Horner. Sin embargo, para multiplicarlos, el algoritmo toma tiempo $\Theta(n^2)$ pues se necesita, por cada coeficiente del primero, realizar la multiplicación con todos los coeficientes del segundo.

Sin embargo, si los polinomios estuvieran en una representación de puntos y evaluación, la multiplicación se obtiene en $\Theta(n)$ pues basta con multiplicarlos término a término. Entonces el problema consiste en intercambiar las representaciones de los polinomios de manera rápida (más rápida en tiempo de cómputo que la multiplicación en la representación por coeficientes). Utilizando la transformada rápida de Fourier se pueden intercambiar la representación de polinomios en $\Theta(n \log\ n)$.

Si se logra lo anterior la multiplicación de dos polinomios en representación de coeficientes se puede obtener de la suguiente manera:

1. Pasar los polinomios a su representación de (punto,evaluación): $\Theta(n \log\ n)$
2. Multiplicar los polinomios en esta nueva representación: $\Theta(n)$
3. Regresar el resultado a la representación de coeficientes: $\Theta(n \log\ n)$

Por lo que la multiplicación toma $\Theta(n \log\ n)$. A continuación se presentan algunos conceptos para obtener el intercambio de las representaciones en el orden dicho anteriormente.

5.2.1. Raíces n-ésimas de la unidad.

Lo primero a tener en cuenta es en qué puntos se debe evaluar el polinomio para poder intercambiar su representación. Hay unos puntos en particular que sirven bastante bien, estos son conocidos como *las raíces de la unidad*.

Dado un polinomio de grado n, hay exactamente n números complejos ω tales que $\omega^n = 1$. A decir: $e^{2\pi ik/n}$ con $k = \{0, \ldots, n-1\}$ e $i = \sqrt{-1}$. Cabe resaltar que $\omega_n = e^{2\pi i/n}$ es *la raíz principal de la unidad* y las demás son potencias de ω_n.

Obsérvese que si se evalua al cuadrado las n raíces n-ésimas de la unidad, cuando n es par, obtenemos cada una de las $n/2$ $(n/2)$-ésimas raíces de la unidad exactamente dos veces. Esto servirá para diseñar el algoritmo secuencial.

Dado que la multiplicación de un polinomio de grado n con uno de grado m puede resultar en un polinomio de grado $n+m$, entonces se debe considerar que si n es el grado mayor, el polinomio de grado m se debe completar con coeficientes iguales a cero para las evaluaciones, y más aún, se deben efectuar $2n$ evaluaciones para que las representaciones tengan $2n$ puntos y se puedan multiplicar correctamente.

Ahora, para pasar de la representación de parejas $(punto, valor)$ a la representación de coeficientes, se utiliza el mismo procedimiento, conocido como la *transformada inversa discreta de Fourier*. El algoritmo para esto se presenta en la siguiente sección.

Considerando todo lo anterior se presenta el siguiente resultado:

Teorema 5.2.1 (Teorema de convolución.)**.** *Sean a y b dos vectores de coeficientes de grado n, donde n es una potencia de dos. Los vectores a y b han sido llenados con ceros hasta la longitud de $2n$. Entonces se da la igualdad:*

$$a \otimes b = DFT_{2n}^{-1}(DFT_{2n}(a) \cdot DFT_{2n}(b))$$

Donde $\cdot$ representa la multiplicación entrada por entrada.

5.3. Un algoritmo secuencial.

Se presenta el algoritmo recursivo que calcula la transformada rápida de Fourier. Lo que se quiere es evaluar un polinomio de grado n, n veces y se desea evaluar en las n raíces n-ésimas de la unidad, por lo que la evaluación de un polinomio con sus n coeficientes: $\overline{a} = \{a_0, \ldots, a_{n-1}\}$ en el valor x está dada por:

$$\overline{a}(x) = \sum_{j=0}^{n-1} a_j x^i$$

Definamos y_k, para $k = 0, \ldots, n-1$ por:

$$y_k = \overline{a}(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

El vector $\overline{y} = \{y_0, \ldots, y_{n-1}\}$ es la *transformada discreta de Fourier* de $\overline{a}$ y contiene las n evaluaciones de $\overline{a}$ en las n raíces n-ésimas de la unidad.

El algoritmo que calcula a este vector utiliza la técnica *divide y vencerás* separando a $\overline{a}$ en dos vectores formados por sus coeficientes de índice par y sus coeficientes de índice impar creando dos polinomios de grado $n/2$:

$$\overline{a}^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}$$
$$\overline{a}^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$

A partir de esto definimos:

$$\overline{a}(x) = \overline{a}^{[0]}(x^2) + x\overline{a}^{[1]}(x^2)$$

Entonces el problema de evaluar $\overline{a}(x)$ en $\omega_n^0, \omega_n^1, \ldots, w_n^{n-1}$ se reduce a:

1. Evaluar los polinomios de grado $n/2$ $\overline{a}^{[0]}(x)$ y $\overline{a}^{[1]}(x)$ en los puntos: $(\omega_n^0)^2, (\omega_n^1)^2, \ldots (\omega_n^{n-1})^2$.
2. Combinar los resultados de acuerdo a la ecuación anterior.

Gracias a las propiedades de las raíces n-ésimas de la unidad, en particular a la observación de la sección anterior, la lista de valores del punto 1 no son n valores distintos, sino solo las $n/2$ raíces $n/2$-ésimas de la unidad, donde cada una aparece exactamente dos veces, por lo que se puede evaluar recursivamente $\overline{a}^{[0]}$ y $\overline{a}^{[1]}$ de grado $n/2$ en las $n/2$ raíces complejas de la unidad. Estos subproblemas tienen la misma forma que el problema original pero la mitad del tamaño. Esto es la base para el siguiente algoritmo recursivo y la elección de estos puntos de evaluación permite obtener la complejidad deseada. El algoritmo recursivo se presenta en el Algoritmo 5.1.

A partir del algoritmo y de las observaciones del párrafo anterior se demuestra que el tiempo de ejecución del algoritmo es dos veces el tiempo de ejecución con una entrada de la mitad del tamaño de la original más un proceso que toma tiempo lineal con respecto al tamaño de la entrada, por lo que el tiempo de ejecución para una entrada de tamaño n, $T(n)$ es $2T(n/2) + \Theta(n) = \Theta(n \log n)$.

Cormen et al. presentan un algoritmo iterativo para implementar la transformada rápida de Fourier de un vector de longitud 2^n y que utiliza una técnica para acomodar los factores que se desean combinar en el orden deseado, [9]. Esta técnica, llamada en inglés *bitreversal permutation*, consiste en dado un número n revertir su representación binaria para crear otro número.

Para dividir el problema de obtener la transformada rápida de Fourier de un vector de tamaño 2^n se requiere calcular las transformadas de los vectores formados por los coeficientes pares e impares, creando así dos subproblemas de tamaño 2^{n-1}. Al considerar el vector de entrada (que son coeficientes de un polinomio) y dividirlo de manera recursiva como se ha mencionado, se obtiene que la posición final en el árbol de recursión del coeficiente i es justamente la permutación revertida a nivel de bits de i. Así que para obtener el algoritmo iterativo primero se deben calcular las posiciones finales de los coeficientes como hojas del árbol, por lo que se presenta el Algoritmo 5.2.

El Algoritmo 5.2 utiliza la función *rev* que calcula la reversa de la representación binaria de k y la transforma en un número. Cabe señalar que como n es una potencia de dos, entonces

Algoritmo 5.1 Transformada rápida de Fourier recursiva.

Entrada $\overline{a}$ es un vector de coeficientes de tamaño una potencia de dos.
Salida $\overline{A}$ contiene la transformada rápida de Fourier de a.

function RecursiveFFT($(\overline{a}, \overline{A})$)
  $n = \overline{a}.length$
  if $n == 1$ **then**
    return $\overline{a}$.
  end if
  $\omega_n = e^{2\pi i/n}$
  $\omega = 1$
  $\overline{a}^{[0]} = (a_0, a_2, \ldots, a_{n-2})$
  $\overline{a}^{[1]} = (a_1, a_3, \ldots, a_{n-1})$
  $\overline{y}^{[0]}$ = RecursiveFFT($\overline{a}^{[0]}$)
  $\overline{y}^{[1]}$ = RecursiveFFT($\overline{a}^{[1]}$)
  for $k = 0$ **to** $n/2 - 1$ **do**
    $y_k = y_k^{[0]} + \omega y_k^{[1]}$
    $y_{k+(n/2)} = y_k^{[0]} - \omega y_k^{[1]}$
    $\omega = \omega\omega_n$
  end for
  return $\overline{y}$
end function

Algoritmo 5.2 Bit-Reverse-Copy.

Entrada $\overline{a}$ es un vector de coeficientes.
Salida En el vector A se alojan los mismos coeficientes pero con su posición permutada por medio de la reversión de su representación binaria.

1: **function** Bit-Reverse-Copy($(\overline{a}, A)$)
2: $n = \overline{a}.length$
3: **for** $k = 0$ **to** $n - 1$ **do**
4: $A[rev(k)] = a_k$
5: **end for**
6: **end function**

todas las evaluaciones de rev son menores que n por lo que el algoritmo funciona bien y tiene complejidad $O(n \log\ n)$. Usando esta función se tiene un algoritmo secuencial e iterativo que calcula la transformada rápida de Fourier de un vector de coeficientes, evaluando en las raíces de la unidad. El algoritmo se presenta en el Algoritmo 5.3.

Algoritmo 5.3 Transformada rápida de Fourier secuencial e iterativa.

Entrada $\overline{a}$ es un vector de coeficientes de tamaño una potencia de dos.
Salida $\overline{A}$ contiene la transformada rápida de Fourier de a.

```
function IterativeFFT((ā, Ā))
    Bit-Reverse-Copy(ā, Ā)
    n = ā.length
    for s = 1 to log(n) do
        m = 2^S
        ω_m = e^{2πi/m}
        for k = 0 to n − 1 by m do
            ω = 1
            for j = 0 to m/2 − 1 do
                t = ω ∗ Ā[k + j + m/2]
                u = Ā[k + j]
                Ā[k + j] = u + t
                Ā[k + j + m/2] = u − t
                ω = ω ∗ ω_m
            end for
        end for
    end for
    return Ā
end function
```

Finalmente, para obtener la inversa de la transformada rápida de Fourier, es decir, pasar de la presentación de evaluación en las raíces de la unidad a la representación con coeficientes[1], simplemente se cambia el valor de ω_m por $e^{-2\pi i/m}$ y cada elemento de $\overline{A}[k]$ se divide entre n.

Con estos algoritmos y usando el teorema de convolución podemos multiplicar polinomios en $\Theta(n \log\ n)$.

5.4. Estrategia de paralelización.

Antes de poder paralelizar el algoritmo, se debe idear una estrategia de paralelización, esto implica, entre otras cosas, establecer la independencia de tareas y los puntos de sincronización en el algoritmo, así como determinar el grado de paralelismo del mismo.

[1] A este proceso se le llama *interpolación*.

Para este caso de estudio, la estrategia fue adaptar el algoritmo secuencial a un algoritmo paralelo preservando la estructura del mismo pero explotando la independencia de tareas para que sean paralelizadas.

El primer punto es observar que el algoritmo secuencial presenta varias etapas. Cada etapa equivale a una etapa de la gráfica de dependencias del problema y donde cada etapa equivale a una iteración del primer *for* del algoritmo secuencial e iterativo. Este *for* se ejecuta un número logarítmico de veces por lo que el algoritmo tiene un número logarítmico de etapas. Claramente, cada etapa es dependiente a la anterior pues se necesita la anterior para calcular la etapa actual, por lo que no se puede paralelizar el cálculo de etapas.

Sin embargo, en una etapa se calcula el vector de evaluaciones correspondientes tomando los elementos dos a dos y a partir del algoritmo se observa que en cada iteración del *for* más interno solo se utilizan y modifican esas dos entradas y el segundo *for* itera sobre los diferentes subproblemas de la etapa. Al inicio los subproblemas tienen tamaño 2, luego tienen tamaño 4 y así hasta que solo hay un subproblema de tamaño n. Estos dos *for* se pueden paralelizar, pues cada subproblema es independiente a otro y más aún, dado que los valores solo se modifican de dos en dos, si el problema tiene tamaño $2k$ (hay que recordar que como n es una potencia de dos, todos los subproblemas tienen siempre tamaño par) entonces se pueden usar k hilos para calcular el resultado en paralelo.

Estas observaciones permiten crear un algoritmo en donde el número de hilos utilizados en cada etapa es constante y no decae conforme las etapas avanzan. En otros problemas el número de hilo decae conforme aumentan las etapas. A este fenómeno le llamamos *pérdida de paralelismo*.

En la siguiente sección se presenta el código en CUDA C de este algoritmo paralelo.

5.5. Algoritmo paralelo en CUDA C.

En el Código 5.1 se presenta el código (sin comentarios) del algoritmo en CUDA C que calcula la transformada rápida de Fourier de un conjunto de elementos ya ordenados por su permutación reversa de bits.

Las últimas líneas del Código 5.1 representan la parte del anfitrión y son las que iteran sobre el número de etapas. En cada etapa se lanza un *kernel* y este es el que procesa una etapa con la estrategia mencionada arriba. El argumento `dev_a` es el arreglo en el dispositivo donde se guardará el resultado. El argumento `pot` es la potencia de dos con la que se está trabajando, la `i` es la etapa actual y el `1` indica si se está trabajando con la transformada o con su inversa (los algoritmos son casi idénticos, salvo por el exponente de ω)

En cuanto al *kernel*, como un hilo puede potencialmente manejar más de dos elementos, las primeras líneas calculan el número de identificador del hilo (`tid`) y el número total de hilos en ejecución (`stride`). Además, se calcula el número de subproblema que el hilo manejará y su desplazamiento interno en dicho subproblema, es decir, si la etapa resuelve vectores de tamaño 4 utilizando la solución de los subproblemas de tamaño 2, entonces hay $2^{pot}/4$

Código 5.1: FFT en CUDA C

```
__global__ void etapaFFT(complejo* A, int pot,
int tam_bloque, int inversa){
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  int numbloque = 2*tid/tam_bloque;
  int numbloqueinterno = tid - numbloque *tam_bloque/2;
  int n = 1<<pot;
  while(tid<n/2){
    complejo omega;
    omega.r = omega.i = 0;
    complejo op1 = A[numbloque*tam_bloque + numbloqueinterno];
    complejo op2 = A[numbloque*tam_bloque + numbloqueinterno
    + tam_bloque/2];
    float arg = inversa?(-2*PI*numbloqueinterno/tam_bloque)
    :(2*PI*numbloqueinterno/tam_bloque);
    omega.r = cos(arg);
    omega.i = sin(arg);
    complejo prod = omega * op2;
    A[numbloque*tam_bloque + numbloqueinterno] = op1 + prod;
    A[numbloque*tam_bloque + numbloqueinterno +
    tam_bloque/2] = op1 - prod;
    tid += stride;
    numbloque = 2*tid/tam_bloque;
    numbloqueinterno = tid - numbloque *tam_bloque/2;
  }
}

for(i = 2; i<= n;i*=2)
        etapaFFT<<<NB,TPB>>>(dev_a,POT,i,1);
```

subproblemas y hay que determinar sobre cuál trabajará el hilo actual y sobre qué pareja de elementos de ese subproblema dado.

El `while` es necesario cuando hay mas elementos que hilos, en particular si $2^{pot} > 2*stride$ es necesario iterar pues hay hilos que necesitaran procesar más de una pareja para completar la etapa. Para procesar la pareja el procedimiento es exactamente el mismo que en el secuencial, de hecho, hay varias líneas que son exactamente iguales.

Cabe señalar que este código supone que `dev_a` ya contiene el resultado de la función *bit reverse copy* y dicha función se puede paralelizar de manera muy sencilla por lo que se omite el código para la misma.

Como se observa a partir del código, en todas las etapas se lanzan el mismo número de hilos y de bloques y más aún, todos los hilos en ejecución hacen trabajo a menos que haya menos elementos que procesar que hilos.

En la siguiente sección se presentan los resultados de ejecuciones obtenidos al comparar esta implementación en CUDA C con la implementación en la versión secuencial.

5.6. Resultados y csonclusiones.

Las pruebas de rendimiento en CPU fueron realizadas en una computadora con un procesador Intel Xeon E5-2680v2 con velocidad de 2.8 GHz, 10 núcleos y 25 MB en caché L3. En el mismo equipo fueron realizadas las pruebas en GPU utilizando una tarjeta GTX 780 de NVIDIA que cuenta con 2304 núcleos CUDA y 3072 MB en memoria RAM.

Las opciones de compilación para CPU fueron: `g++ -O3` mientras que las de nvcc fueron: `nvcc -O3 -arch=sm_30`. Cada programa se ejecuto 100 veces y se midió el tiempo en milisegundos que tardó en ejecutar solo el procedimiento para calcular la transformada rápida de Fourier. A partir de las ejecuciones se calculó el promedio. En la Figura 5.1 se muestra la comparación de rendimiento entre la versión secuencial y la versión paralela. Así mismo, en la Figura 5.2 se muestra el factor de aceleración de la versión paralela contra la versión secuencial. Este factor se obtiene simplemente dividiendo el tiempo de la versión paralela entre el tiempo de la versión secuencial para cada tamaño de entrada.

La conclusión obtenida a partir de estos resultados, es que este algoritmo se presta muy bien para ser realizado en CUDA® y todo se debe a que nunca se pierde paralelismo en los hilos. Cabe señalar que el algoritmo presentado no se optimizó, sino que se le dejó esa tarea al compilador. Una optimización sería realizar las primeras etapas en memoria compartida para tratar de mejorar aún más el rendimiento.

La etapa de dividir el problema consiste en calcular las permutaciones de bits, otra optimización sería paralelizar esto, que como se ya se dijo, no es muy difícil.

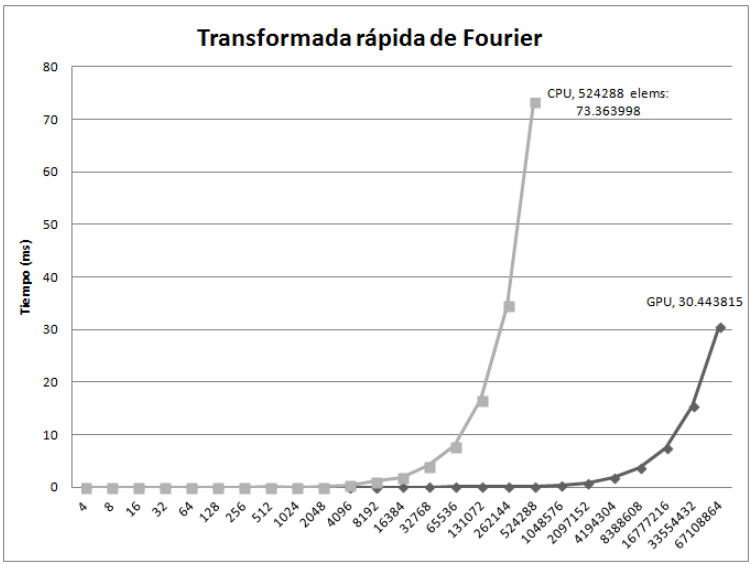

Figura 5.1: Comparativa de rendimiento de la transformada rápida de Fourier.

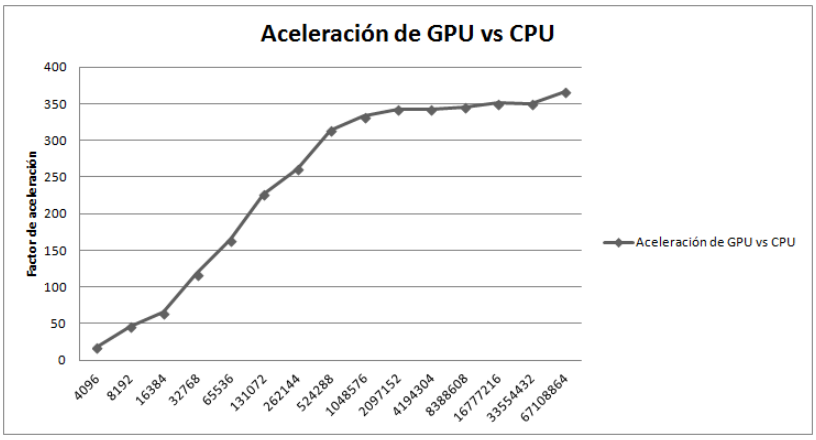

Figura 5.2: Gráfica de aceleración de la transformada rápida de Fourier.

Capítulo 6

Caso de estudio: subrutina de mezcla.

En este capítulo se presenta la subrutina de mezcla que se utiliza en el algoritmo de ordenación *mergesort*. Esta subrutina se encuentra en un capítulo aparte porque a pesar de que se puede implementar de manera secuencial, existe una manera de paralelizarla no intuitiva y que presenta un gran comportamiento en CUDA.

En el desarrollo de este caso de estudio se implementó una estrategia para crear un algoritmo que simule el paralelismo dinámico en CUDA. Esto se explica con mayor detalle en la sección 6.4.

6.1. Motivo de estudio.

El motivo por el que se eligió la subrutina de *merge* fue porque desde el inicio se escogió el algoritmo de ordenación *mergesort* (presentado en el siguiente capítulo) y Cormen et al. describen una manera de paralelizar la subrutina de *merge* usando *divide y vencerás*, [9].

Además de lo anterior, en el desarrollo del trabajo se utiliza una técnica de paralelización que funciona de manera excelente en CUDA pues explota al máximo tanto la arquitectura como la jerarquía de hilos, bloques y etapas.

6.2. Planteamiento del problema.

La idea detrás del problema se basa en crear una secuencia ordenada de elementos a partir de dos subsecuencias ya ordenadas más pequeñas. Con esta idea básica se observa que el problema en sí suena bastante sencillo y en realidad lo es, pues hay un algoritmo secuencial muy sencillo para resolverlo.

La posibilidad de paralelización surge tras observar que la idea fundamental es crear una secuencia más grande a partir de dos de menor tamaño. Esta idea ha estado presente a lo

largo del trabajo cuando se habla de algoritmos *divide y vencerás*. Sin embargo, en los otros algoritmos había una etapa de vencer, en la que la solución se unía en una más grande. De hecho, la subrutina de *merge* es la etapa de vencer de *mergesort* como se verá en el capítulo siguiente.

Entonces, pensando de abajo hacía arriba, se tienen dos pequeñas secuencias de elementos ordenados (el caso base es una secuencia de un solo elemento, o tal vez de cero elementos) y se quiere mezclarlas para crear una más grande. La idea simple es ir comparando de izquierda a derecha ambas secuencias y el elemento más pequeño es removido de la secuencia correspondiente e insertado en el resultado.

Esta idea provee un algoritmo secuencial bastante eficiente, de hecho, óptimo. Sin embargo, es inherentemente secuencial, es decir, esta idea como tal no se puede paralelizar, pues si hubiera al menos dos hilos comparando concurrentemente los elementos, para que el hilo i calcule la posición final del elemento e_j en la secuencia de salida, necesita saber cuántos elementos hay antes de j por lo que debe de recalcular el trabajo hecho por los otros hilos, por lo que no habría paralelismo real. En los siguientes párrafos se discute una idea alternativa para mezclar dos subsecuencias ordenadas en una secuencia ordenada que contenga a las dos anteriores en donde sí se exhiba una estrategia paralela.

Antes de comenzar con la idea se presenta un pequeño algoritmo de búsqueda en secuencias conocido como *búsqueda binaria*. La idea es la siguiente: se tiene una secuencia $S = \{s_1, s_2, \ldots, s_n\}$ de elementos y un elemento p que se quiere buscar en S. En principio, para contestar la pregunta ¿p está en S? se necesita comparar a p con todos y cada uno de los elementos de S, pues si se omite alguna comparación y se concluye que p no está en S, pudiera darse el caso de que dicho elemento omitido es en efecto p. Es fácil demostrar que el problema de buscar a un elemento en una secuencia tiene complejidad temporal $\theta(n)$ donde n es el tamaño de la secuencia a buscar.

Sin embargo, ¿qué pasa si S está ordenado (suponiendo que los elementos de S son totalmente ordenables)? En este caso la idea cambia, pues si S está ordenado, para responder la pregunta, basta encontrar el primer elemento $s_i \geq p$, pues como S está ordenado, todos los elementos a la derecha de s_i son mayores o iguales a s_i por lo que p no podría estar entre ellos. Ahora si se encuentra al primer $s_j \leq s_i$ tal que $s_j \leq p$, se tendría que $s_{j-1} \leq s_j \leq p \leq s_i \leq s_{i+1}$, por lo que con esto se ha acotado el posible rango de p. Ahora, si se toma a $s_{n/2}$ como comparación inicial para p se tiene que si $p \geq s_{n/2}$ entonces la búsqueda de p se restringe solo a $\{s_{n/2}, \ldots, s_n\}$ y si $p \leq s_{n/2}$ se restringe solo a $\{s_1, \ldots, s_{n/2}\}$. Si se repite este procedimiento de manera recursiva, donde cada vez que el elemento de la mitad sea mayor o igual que p se mantiene la mitad superior de la secuencia y dualmente, cada que el elemento de la mitad sea menor o igual que p se mantiene la mitad inferior de la secuencia, entonces en cada paso el rango de búsqueda disminuye y por la consecuencia de que estén ordenados, en cada paso se desechan elementos que no son p. Obviamente, llegará un momento en donde la secuencia a buscar sea solo de un elemento o sea vacía en cuyo caso se podrá responder con certeza la pregunta.

Al proceso de búsqueda recursivo descrito en el párrafo anterior se le conoce como *búsqueda binaria* y solo se puede utilizar cuando la secuencia a buscar está ordenada linealmente[1].

[1]En matemáticas, un orden lineal o total es aquel donde la relación de orden $\leq$ cumple con la ley de

Este proceso tiene complejidad temporal $\theta(\log\ n)$ que es asintóticamente mas bajo que el de búsqueda lineal. Este algoritmo es ampliamente usado en estructuras de datos ordenadas para buscar elementos en ellas.

Una vez descrito el proceso de búsqueda, sean dos secuencias ordenadas de elementos $A = \{a_1, a_2, \ldots, a_n\}$ y $B = \{b_1, b_2, \ldots, b_m\}$ que se quieren mezclar para obtener una secuencia con todos los elementos de A y de B en orden. Considérese la siguiente observación: como A está ordenado, la mediana de A es por definición, el elemento a la mitad de A; es decir $a_{n/2}$. Bien, ¿qué pasa si este elemento es buscado en B utilizando *búsqueda binaria*? Una vez que se ha encontrado al primer elemento b_i tal que $a_{n/2} \leq b_i$, se puede partir a A y a B en cuatro subsecuencias $A_1 = \{a_1, \ldots, a_{(n/2)-1}\}$, $A_2 = \{a_{n/2}, \ldots, a_n\}$, $B_1 = \{b_1, \ldots, b_{i-1}\}$, $B_2 = \{b_i, \ldots, b_m\}$.

Por la forma en que las subsecuencias anteriores fueron creadas cumplen que todos los elementos de A_1 son menores o iguales que b_i, por lo que son menores o iguales que todos los elementos de B_2, análogamente, todos los elementos de B_1 son menores o iguales que los elementos de A_2 pues b_i es el primer elemento de B mayor o igual que la mediana de A. Entonces, a partir de esta observación se concluye que si se mezclan primero las subsecuencias A_1 y B_1, todos estos elementos serán menores o iguales que todos los elementos en la mezcla de A_2 y B_2 por lo que no importa qué mezcla se realice primero, es decir, la mezcla de A_1 con B_1 es una tarea independiente a la mezcla de A_2 con B_2. Con esta interesante manera de mezclar se exhibe una independencia de tareas, la cual por supuesto, puede ser paralelizada. Este será el algoritmo de mezcla que se paralelizará e implementará en CUDA C, aunque con un nivel de paralelismo aún más fino.

6.3. Un algoritmo secuencial.

En esta sección se muestra un algoritmo secuencial para el problema de la mezcla de subsecuencias ordenadas. Este algoritmo es el descrito al final de la sección anterior y utiliza búsqueda binaria para hacer explícita la independencia de tareas. El algoritmo se encuentra en el Algoritmo 6.1.

En el algoritmo la operación $\cdot$ denota la concatenación de secuencias. La función *binarySearch* implementa la *búsqueda binaria*.

6.4. Estrategia de paralelización.

Dado que el algoritmo presenta el paradigma *divide y vencerás* se deben identificar las tres etapas. La etapa de vencer corresponde claramente con las llamadas recursivas y la etapa de combinar es la última línea. Mientras que la etapa de dividir consiste en construir los conjuntos A_1, A_2, B_1 y B_2. Sin embargo, para obtener esta división es necesario calcular la posición de la mediana de A en B. El problema con este algoritmo es que esa posición no se puede obtener en tiempo de compilación pues si se quiere utilizar esta subrutina de

tricotomía.

Algoritmo 6.1 Algoritmo secuencial de mezcla con búsqueda binaria.

Entrada $A = \{a_1, a_2, \ldots, a_n\}$ y $B = \{b_1, b_2, \ldots, b_m\}$ dos secuencias ordenadas de elementos.
Salida La secuencia ordenada C formada por los elementos de A y de B.

```
function RecursiveMergeBB(A, B)
    if A.length == 1 then
        pos_B = binarySearch(a_1, B)
        return B[0, ..., pos_B] · A · B[pos_B ..., m]
    end if
    if B.length == 1 then
        pos_A = binarySearch(b_1, A)
        return A[0, ..., pos_A] · B · B[pos_A ..., n]
    end if
    pos_B = binarySearch(a_{n/2}, B)
    A_1 = {a_1, ..., a_{(n/2)-1}}; A_2 = {a_{n/2}, ..., a_n}
    B_1 = {b_1, ..., b_{pos_B - 1}}; B_2 = {b_{pos_B}, ..., b_m}.
    C = RecursiveMergeBB(A_1, B_1) · RecursiveMergeBB(A_2, B_2)
    return C
end function
```

mezcla para ordenar los números con *mergesort*, dada la entrada no podemos saber cuál es la posición de la mediana de A en B hasta haber ordenado tanto a A como a B. Este algoritmo en particular presenta paralelismo no precalculable y la etapa de división no es sencilla.

El problema con el paralelismo no precalculable es que se desconoce el tamaño ni los elementos que pertenecen a las etapas paralelas y esto lleva a un gran problema para la implementación en CUDA C pues para hacerlo se necesita algo llamado paralelismo dinámico.

El *paralelismo dinámico* es una propiedad de la arquitectura paralela donde se pueden lanzar y terminar hilos a voluntad del proceso mediante llamadas al sistema o procedimientos remotos. Como un ejemplo de esto, en sistemas operativos modernos, se tiene el procedimiento *fork* para que un hilo de proceso cree otro hilo que depende de él. El paralelismo dinámico es muy versátil pero usualmente el lanzar y mezclar (mediante funciones como *join*) hilos requiere de un alto grado de instrucciones administrativas, por lo que no es recomendable tener grandes cantidades de hilos a la vez.

Hasta el momento del desarrollo de este trabajo el paralelismo dinámico no era muy eficiente en la arquitectura y de hecho, hasta antes de la capacidad de cómputo 3.5 no existía el paralelismo dinámico en CUDA.

En el Algoritmo 6.2 se presenta el pseudocódigo para la subrutina *merge* que utiliza la estrategia de *búsqueda binaria* y paralelismo dinámico para las llamadas recursivas. En el algoritmo, el procedimiento *BinarySearch* toma cuatro argumentos, el primero es el elemento a buscar, el segundo es el arreglo donde se buscará y los últimos dos son los índices de inicio y final del arreglo en donde se buscará al elemento. Esto para indicar exactamente en qué región del arreglo se debe buscar al elemento.

Por otro lado, la función $p - merge$ recibe: el arreglo de donde se tomarán los subarreglos a mezclar, el inicio y el final del primer subarreglo, el inicio y el final del segundo subarreglo,

Algoritmo 6.2 *Merge* recursivo en paralelo

Entrada El algoritmo mezcla dos subarreglos no necesariamente adyacentes y los guarda en un arreglo de salida. El primer elemento queda en la posición l_3 del arreglo de salida.

Salida El arreglo de salida A que tiene la mezcla de los subarreglos de T indicados por sus índices iniciales y finales.

function $p - merge((T, l_1, r_1, l_2, r_2, A, l_3))$
  $n_1 = r_1 - l_1 + 1$
  $n_2 = r_2 - l_2 + 1$
  if $n_1 < n_2$ **then**
    $swap(l_1, l_2)$; $swap(r_1, r_2)$; $swap(n_1, n_2)$
  end if
  if $n_1 == 0$ **then**
    return
  else
    $q_1 = \lfloor (l_1 + r_1)/2 \rfloor$
    $q_2 = BinarySearch(T[q_1], T, l_2, r_2)$
    $q_3 = l_3 + (q_1 - l_1) + (q_2 - l_2)$
    $A[q_3] = T[q_1]$
    Spawn $p - merge(T, l_1, q_1 - 1, l_2, q_2 - 1, A, l_3)$
    $p - merge(T, q_1 + 1, r_1, q_2, r_2, A, q_3 + 1)$
    Sync
  end if
end function

el arreglo donde se almacenará la mezcla y el índice de inicio de dicha porción del arreglo de resultados, en ese orden.

La estrategia del algoritmo es similar a la versión secuencial: al inicio se asegura que el primer subarreglo esté efectivamente a la izquierda del segundo, sino, se intercambian. Una vez asegurado esto se calcula la posición de la mediana del primero en el segundo y la posición donde se empezará a almacenar la segunda parte de los subarreglos en el arreglo de salida. Finalmente se ejecuta mediante la instrucción paralela *Spawn* la primera llamada recursiva. Esta instrucción lanza un hilo que ejecutará dicha llamada y que cuando termine ejecutará la instrucción *Sync* para entregar sus resultados y terminar su ejecución. Con esto se logra que esta llamada se haga en paralelo e independientemente de la segunda llamada recursiva que ejecuta el mismo hilo que ejecutó las instrucciones anteriores. Para el momento de la ejecución de la instrucción *Sync* ambas subrutinas terminaron su ejecución por lo que ambas partes están mezcladas en A a partir del índice p_3.

6.4.1. Paralelismo dinámico en CUDA C.

Como ya se mencionó, en CUDA no había paralelismo dinámico y de hecho el que hay ahora no se acopla muy bien a este modelo pues solo se permiten lanzar *kernels* anidados. Además, el paralelismo dinámico es necesario en este algoritmo pues como ya se mencionó, este algoritmo cuenta con paralelismo no precalculable. Más aún, el algoritmo debe ser implementado de forma iterativa de abajo hacia arriba y no en forma recursiva como es presentado en el Algoritmo 6.2.

Por lo anterior fue necesario idear una estrategia para simular el paralelismo dinámico de tareas en CUDA C creando una lista de tareas por realizar, donde cada lista de tareas es una tarea básica. En otras palabras se intentó simular la generación del árbol de recursión en tiempo de ejecución.

Algoritmo 6.3 Protocolo de simulación de paralelismo dinámico en CUDA C.

while Subejemplares S no vacio **do**
  Toma una tarea t de S
  Calcular los dos subejemplares recursivos s_1 y s_2
  Si s_1 o s_2 son casos base, colocarlos en el arreglo de hojas.
  si alguna es recursiva, encolarla en S.
  Syncthreads
end while

El protocolo presentado en el Algoritmo 6.3 primero toma una subtarea recursiva. Al inicio del protocolo S contiene al ejemplar recursivo original. Se usa sincronización de barrera entre cada una de las etapas evitar corrupción de datos.

Una vez que el hilo tiene la tarea con la que trabajará, este calcula los dos subejemplares recursivos tal y como en el algoritmo secuencial, es decir, encontrando la mediana de la primera en la segunda y calculando los cuatro subarreglos a mezclar.

Después verifica si alguna de estas subtareas es una "hoja" o no. Una hoja, se refiere a un caso base. Los casos base son aquellos subejemplares de tamaño menor o igual a 1. En este caso, si un subarreglo tiene tamaño 1, entonces se puede calcular su posición final en el *merge*. Si un arreglo es vacío, entonces ya está mezclado con el otro. Para llevar un registro de las tareas base se almacenan en un arreglo.

Por otro lado, si alguna de las dos subtareas es recursiva; es decir, ninguno de sus dos arreglos a mezclar tiene tamaño menor o igual a uno, entonces se encolan en S.

Este protocolo termina pues cada que se elige una tarea, o se reduce el tamaño de la tarea o se encola una "hoja". En cualquiera de los dos casos el arreglo S terminará por vaciarse. Cabe mencionar que se necesitan operaciones atómicas para manipular el arreglo de hojas y el arreglo S para evitar corrupción de datos, sin embargo, en la sección de resultados se menciona otra manera de manipular estos arreglos.

Cabe señalar que este protocolo no fue implementado en CUDA C, pues aunque puede simular el paralelismo dinámico se encontraron otras fuentes en donde se describe otra manera de implementar el algoritmo de *mergesort*, [10, 23].

La idea fundamental para implementar el *merge* en una arquitectura masivamente paralela como CUDA es la importante observación de que en el algoritmo recursivo se conoce con exactitud la posición de la mediana de A en A y al buscarla en B se conoce exactamente cuantos elementos menores o iguales que él hay en B por lo que si se le suma la mitad de elementos menores que él en A se conoce exactamente su posición en el arreglo mezclado. Es decir, siempre se conoce la posición exacta de la mediana

Esta poderosa observación se puede generalizar, pues si se elige un elemento $a_i \in A$ (recordando que $A = \{a_1, a_2, \ldots, a_n\}$ y $B = \{b_1, b_2, \ldots, b_m\}$), como A está ordenado, entonces se conoce cuántos elementos menores o iguales que a_i hay en A, a decir, $i - 1$. Si se encuentra el lugar de a_i en B, es decir, se calcula cuantos elementos hay en B menores o iguales a a_i, a decir j, entonces la posición en el arreglo mezclado de a_i está dada por $i - 1 + j$ por lo que se conoce exactamente la posición de todos los elementos de A en el arreglo resultado. Al hacer lo mismo para los elementos de B, se tiene la posición exacta de todos los elementos en el arreglo a mezclar.

Debido al gran número de hilos en CUDA C esta idea es la que se utilizará para implementar el algoritmo de *merge*. La idea entonces es que cada hilo tome un elemento, determine si es de A o de B y busque su lugar en el otro arreglo mediante *búsqueda binaria*. Finalmente calcula su posición exacta en el arreglo de salida sumando su posición en su arreglo origen más la posición obtenida al buscar el elemento en el otro arreglo.

6.5. Algoritmo paralelo en CUDA C.

En el Algoritmo 6.4 se muestra el pseudocódigo que calcula el *merge* de dos bloques.

Algoritmo 6.4 *Merge* en CUDA C.

Entrada Dos arreglos A y B.
Salida B tiene el resultado de mezclar las dos mitades de A
Calcular si el hilo trabajará con la parte izquierda o derecha de A
Calcular posiciones de trabajo, es decir, los elementos a procesar.
Buscar la posición de los elementos a procesar en el otro bloque y guardar su posición.
A partir de la posición en el bloque de trabajo y la obtenida vía búsqueda binaria en el otro bloque calcular p que es la posición en el arreglo mezclado.
Escribir en $B[p]$ el elemento procesado.

6.6. Resultados y conclusiones.

El protocolo de paralelismo dinámico es generalizable a cualquier algoritmo *divide y vencerás* que presente paralelismo no precalculable o dinámico pues la parte de crear los subejemplares es muy general y se puede definir particularmente para el problema a tratar.

Por otro lado, aunque la manipulación de los arreglos de subtareas y hojas se plantea inicialmente de manera atómica se puede utilizar un algoritmo para compactar dichas estructuras.

Cuando se habla de un arreglo *compacto* se refiere a un arreglo donde hay una serie de entradas válidas e inválidas y el arreglo cumple que a partir de una posición i las entradas antes de i son válidas y las entradas después de i son no válidas. Visto como un arreglo binario, donde 1 representa una entrada válida y 0 una entrada inválida, el arreglo está *compacto* si a partir del primer 0, todos los elementos anteriores son 1 y el resto son 0. La operación de *compactar* es entonces, transformar un arreglo arbitrario en un arreglo compacto. Por supuesto esta es una propiedad *booleana* (valuada en 0 ó 1) por lo que la operación de compactar se puede generalizar de la siguiente manera:

Definición 6.6.1 (Arreglo compacto)**.** *Sea $A = [a_1, \ldots, a_n]$ un arreglo de n elementos de un conjunto S. Sea $P(x)$ una propiedad lógica de los elementos de S. Decimos que A está compacto bajo P si $\forall\ 1 \leq i \leq j \leq n, \neg P(a_i) \rightarrow \neg P(a_j)$.*

Este algoritmo permite que cada hilo escriba en estas estructuras en una posición designada y luego *compactar* este arreglo para que se ocupen solo las posicines iniciales y el resto queden vacías. Cabe mencionar que este algoritmo se implementó con el fin de eliminar la pérdida de rendimiento que ocasionan las operaciones atómicas. Sin embargo, debido a la jerarquía de memoria de CUDA este algoritmo no dio el resultado esperado pues el tiempo necesario para compactar es incluso mayor al utilizado por las operaciones atómicas.

Por este motivo y debido a la información encontrada en otra fuente se utilizó el otro algoritmo, [23] .

En cuanto a los resultados del algoritmo utilizado, dado que esta es solo una parte del algoritmo de *mergesort* presentado en el siguiente capítulo, se decidió medir la ejecución total de *mergesort* y no solo la ejecución de la subrutina de mezcla.

La conclusión más relevante tras haber trabajado en este problema es que se encontró, implementó y probó un algoritmo que explota de sobremanera la arquitectura masivamente paralela de CUDA. Esto llevó a un algoritmo sencillo y elegante, en donde el paralelismo no era nada evidente, pues la subrutina original de mezcla era totalmente secuencial. Más aún, se explotó la idea de la *búsqueda binaria* hasta el punto más fino posible. En lugar de hacer una rutina de búsqueda para subdividir el problema en dos, cada hilo hace su propia búsqueda para calcular su posición final.

En el siguiente capítulo se utilizará esta idea para diseñar un algoritmo *mergesort* masivamente paralelo.

Capítulo 7

Caso de estudio: algoritmo de ordenación por mezcla (*mergesort*).

En este capítulo se presenta el tercer caso de estudio que está íntimamente ligado al caso anterior: El algoritmo de ordenación por mezcla (*mergesort*) es óptimo (su complejidad es óptima) y la mayoría de las implementaciones producen un algoritmo estable[1] lo que lo convierte en uno de los algoritmos de ordenamiento más conocidos y utilizados.

7.1. Motivo de estudio.

El algoritmo *mergesort* fue inventado por uno de los padres de la computación: el matemático húngaro *John Von Neumann* en 1945 y el algoritmo fue diseñado mediante el paradigma *divide y vencerás*.

Como ya se ha mencionado, este algoritmo es muy utilizado. De hecho, es tal vez, el más utilizado por las bibliotecas y API que tengan una función de ordenar, tal vez solo compite con *quicksort* y la razón es que *quicksort* es más fácil y rápido de implementar y en ejemplares de trabajo pequeñas funciona bien a pesar de que su complejidad no es óptima.

La razón por la que se quiere paralelizar *mergesort* es porque la solución a varios problemas computacionales requiere ordenar elementos (el caso del cierre convexo, optimización de recursos, entre otros.). En otras palabras, si en muchos problemas se necesita ordenar, ¿por qué no intentar mejorar el tiempo requerido para ello?

7.2. Planteamiento del problema.

El algoritmo de ordenación por mezcla (se usará el término en inglés *mergesort* de manera indistinta) ordena una secuencia de elementos totalmente comparables entre sí utilizando la

[1] Estable significa que si en la secuencia original hay dos elementos iguales, digamos a y b, y a aparece antes que b, entonces en la secuencia ordenada a aparecerá antes que b.

subrutina de mezcla (*merge*) vista en el capítulo anterior.

Esta subrutina es utilizada en cada paso de recursión. De hecho, es la etapa de combinar en la estructura del paradigma *divide y vencerás*. La etapa de dividir en este caso es trivial pues solo se necesita separar al arreglo original a la mitad. Una vez separado, ambas mitades se ordenan recursivamente lo que equivale a la etapa de vencer. Cuando ambas mitades están ordenadas se combinan utilizando la subrutina de *merge* formando así una secuencia ordenada de elementos.

7.3. Un algoritmo secuencial.

El algoritmo secuencial recursivo es muy sencillo y claro a partir de lo ya discutido y en particular, al utilizar la subrutina de *merge* del capítulo anterior. El algoritmo se presenta en el Algoritmo 7.1.

Algoritmo 7.1 *Mergesort* Secuencial.

Entrada A es la secuencia a ordenar, p es el inicio de la subsecuencia de A a ordenar y r el final. En la primera llamada del algoritmo p es 0 y r es el tamaño de A.
Salida A queda ordenada.

```
function MERGESORT(A, l, r)
    if p < r then
        q = ⌊(l + r)/2⌋
        mergesort(A, l, q)
        mergesort(A, q + 1, r)
        A = RecursiveMergeBB(A[l..q], A[q + 1..r])
    end if
end function
```

La primera observación es que el algoritmo modifica al conjunto de entrada A. Sin embargo, cuando se utiliza un algoritmo con aproximación de abajo hacia arriba como lo es en el caso de CUDA C usualmente se utilizan dos búffers para almacenar los datos. En el primero se guardan los datos ordenados hasta el momento (en un principio este búffer tendría a la secuencia original) y en el segundo se almacenan los resultados de la etapa actual. Al finalizar la etapa actual estos dos búffers se intercambian conceptualmente, siendo ahora el segundo la secuencia de entrada para la siguiente etapa. A esta técnica se le conoce como *doble búffer* y es la que se utilizará en la implementación de *mergesort*.

7.4. Estrategia de paralelización.

En el desarrollo del trabajo se crearon dos algoritmos en CUDA C que implementaban *mergesort*. Ambos, por supuesto, implementan el algoritmo de abajo hacia arriba y de manera iterativa.

Primera implementación.

En esta implementación primero se ordenan bloques de tamaño dos, luego se toman dos bloques de tamaño dos que se mezclan para formar uno de tamaño cuatro y estos a su vez, se mezclan para formar bloques ordenados de tamaño ocho y se procede de esta manera hasta obtener un solo bloque ordenado del tamaño del conjunto de entrada. La mezcla en este algoritmo se realiza de manera secuencial tal y como la mezcla original del algoritmo. Cabe mencionar que aunque se establece la idea pensando que el tamaño del arreglo es una potencia de dos, utilizar la idea para arreglos de tamaño arbitrario requiere unas modificaciones muy sencillas.

Si la secuencia a ordenar tiene tamaño n, entonces en la primera etapa se necesitan $n/2$ hilos para realizar la mezcla y formar bloques de tamaño 2. Sin embargo, para la segunda etapa solo se necesitan $n/4$ hilos, pues solo hay $n/2$ bloques de tamaño 2 y la mezcla es secuencial por lo que por cada dos bloques de estos solo se puede utilizar un hilo para mezclarlos. Continuando de esta manera en cada etapa siguiente se necesitan exactamente la mitad de hilos que en la etapa actual, por lo que para la última etapa (cuando ahora solo hay dos bloques en total para mezclar) se puede utilizar solo un hilo.

Es evidente que esta estrategia resuelve el problema y de hecho no suena tan mal en un principio pues es claro que en las primeras etapas se tienen muchas mezclas en paralelo y de hecho, aún la última etapa es exactamente igual a la última etapa en un algoritmo secuencial (equivale a hacer la mezcla en la primera llamada a la función en un algoritmo recursivo).

Segunda implementación.

Sin embargo, como ya se describió en el capítulo anterior, utilizando búsqueda binaria se puede paralelizar la etapa de mezcla y al combinar la estrategia con la arquitectura de hilos masiva en CUDA se puede obtener un algoritmo en donde cada hilo procesa un número fijo de elementos y busca su lugar en el arreglo final mediante búsqueda binaria.

Dado que esta estrategia resuelve el problema de mezclar las subsecuencias aún se necesita una serie de etapas (de hecho, se necesitan un número logarítmico de etapas con respecto a n) para ordenar todos los números. La estrategia del algoritmo es entonces utilizar un número logarítmico de etapas. En cada etapa se utiliza la subrutina paralela de *merge* mediante búsqueda binaria para mezclar dos subsecuencias contiguas en una más grande. Este procedimiento continua hasta obtener una sola secuencia mezclada del tamaño del arreglo de entrada. En ese momento termina la ejecución del algoritmo y se obtiene el resultado deseado.

La diferencia, con respecto a la primera implementación, radica en la forma en que se hace la mezcla pues si cada hilo procesa solo dos elementos (digamos uno de la primera subsecuencia y el otro de la segunda), no importa de qué tamaño sean las subsecuencias, ya que en cada etapa se necesitan exactamente el mismo número de hilos para procesar todas las subsecuencias, en particular, en este caso se necesitan $n/2$ hilos activos considerando n como el tamaño de la entrada.

Código 7.1: *Mergesort* en CUDA C.

```
for(i=EPB;i<elems;i<<=1,reps<<=1){
merge<<<bloques,hilos>>>(d_array,d_array_s,elems,i,reps);
swap(d_array_s,d_array);
}
```

Con esta nueva estrategia se utiliza siempre el mismo número de hilos evitando así el efecto de pérdida de paralelismo que se tenía en la estrategia anterior. En la siguiente sección se presenta el código de esta estrategia implementado en CUDA C y se dan más detalles de la implementación.

7.5. Algoritmo paralelo en CUDA C.

En esta sección se muestra el algoritmo de *mergesort* implementado en CUDA C. Para la implementación se necesita un *kernel* que haga la mezcla en cada etapa del árbol de recursión y se necesitan tantos *kernels* como niveles tenga dicho árbol.

El Código 7.1 muestra el código en CUDA C del algoritmo *mergesort*. En el código, `i` representa el tamaño de cada subsecuencia a ordenar, `elems` es el número total de elementos y `reps` lleva el conteo de cuántas repeticiones se han hecho, esto debido a que se usa la técnica de *doble búffer* descrita anteriormente y se debe saber en qué búffer quedaron los resultados finales. Las líneas después del lanzamiento del *kernel* intercambian los búffers. El *kernel* es exactamente la subrutina de *merge* presentada el capítulo anterior.

El código del *kernel* se omite pues es la misma subrutina de *merge* descrita en la sección 6.5. Existe un *kernel* que se ejecuta antes del `for` presentado anteriormente. Este *kernel* se encarga de procesar las primeras etapas del algoritmo, es decir, las etapas en donde las subsecuencias a procesar son muy pequeñas y hay muchas subsecuencias pequeñas que se pueden ejecutar en paralelo en un mismo bloque utilizando memoria compartida. El código realiza el mismo procedimiento de mezcla solo que lo hace a nivel de bloque en un mismo búffer utilizando memoria compartida. En este código no se utiliza la técnica de *doble búffer* sino sincronización a nivel de bloque mediante `__syncthreads()`.

7.6. Resultados y conclusiones.

Las pruebas fueron realizadas en el mismo equipo en el que se realizaron las del programa de la transformada rápida de Fourier descrita en la sección 5.6. Para este caso de estudio se realizaron tres pruebas, la primera fue la versión secuencial, la segunda la versión en CUDA C realizando todas las etapas mediante *kernels* y la tercera fue la versión donde las primeras etapas se realizaron en un solo *kernel* utilizando memoria compartida y sincronización de bloque. El cálculo de los tiempos se hizo ejecutando cada versión 100 veces y obteniendo el promedio. En la gráfica 7.1 se muestra la comparación de rendimiento entre las tres

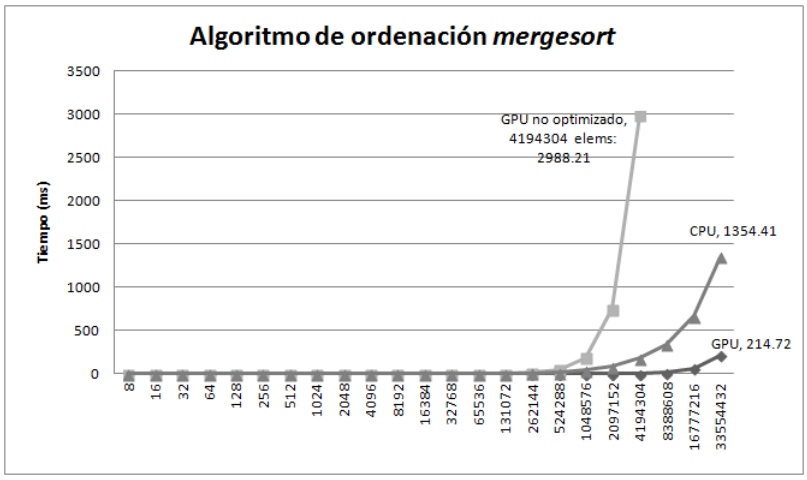

Figura 7.1: Comparativa de rendimiento del algoritmo de ordenación por mezcla.

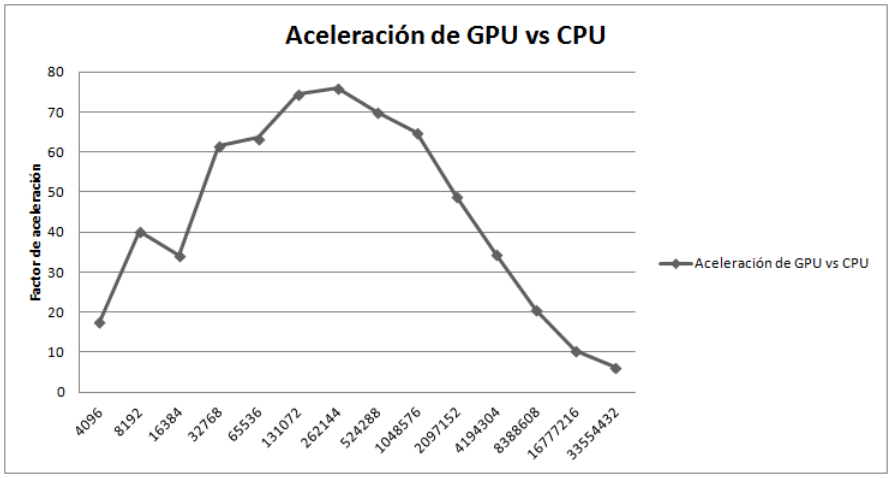

Figura 7.2: Gráfica de aceleración del algoritmo de ordenación por mezcla.

versiones.

En la Figura 7.1 se observa que la versión no óptima (la que realiza todas las etapas en *kernels* separados) realmente no mejora el rendimiento en comparación a la versión secuencial. De hecho, es mucho más ineficiente. Sin embargo, la versión optimizada en CUDA C; es decir, la que utiliza memoria compartida, sí mejora el rendimiento de la versión secuencial.

En la Figura 7.2 se muestra la mejora obtenida entre la versión secuencial y la versión optimizada en CUDA C con respecto al tamaño de cada entrada. De nuevo, estos valores se calcularon dividiendo el tiempo de la versión secuencial entre el tiempo de la versión optimizada en CUDA C.

La mejora obtenida por la versión en memoria compartida muestra que este tipo de memoria sigue siendo muy benéfico para muchas aplicaciones en CUDA ya que esta memoria reside en el chip, en particular, es la misma utilizada por el caché L1, por lo que se dice que es memoria caché a disposición del programador. La desventaja de esta memoria, como se ha hecho notar en el capítulo tres, es que es limitada. Esto restringe un poco su uso; sin embargo, se puede

configurar para obtener 48 Kilobytes de memoria compartida, que al guardar enteros, como en este caso, se convierten en más de 12 mil enteros de 32 bits por bloque en memoria compartida, lo cual es bastante.

También, se concluye que a veces no basta con adecuar la implementación del algoritmo a CUDA C, sino que se tiene que pensar cómo utilizar de mejor manera los recursos disponibles. En particular los distintos tipos y jerarquías de memoria, cosa que no se obtiene a partir del algoritmo. Es decir, tal como sucede en el cómputo secuencial, la implementación final depende totalmente de la arquitectura y recursos disponibles. En este caso, la implementación burda del algoritmo resulta en un programa mucho menos eficiente que la versión secuencial llevando a pensar que tal vez, dicho algoritmo no es bueno[2] para ser implementado en CUDA C. Al haber hecho la optimización (que en realidad no modifica el algoritmo paralelo), se logra una mejora significativa sobre la versión secuencial reivindicando la estrategia de solución del problema.

[2]Bueno en términos de mejora sobre la versión secuencial.

Capítulo 8

Caso de estudio: la pareja de puntos más cercana.

Este es el último caso de estudio presentado en este trabajo y en él se resuelve computacionalmente un problema geométrico.

El problema consiste en: dada una colección de puntos en el plano, encontrar a la pareja cuya distancia *euclidiana* sea la menor. Es decir, encontrar a la pareja de puntos que se encuentren más cerca uno del otro.

Durante el desarrollo de este caso de estudio se utilizó una implementación de un algoritmo paralelo muy interesante que se presenta en la sección de estrategia de paralelización. Este algoritmo es conocido en inglés como *reduce* o *fold* y es un algoritmo muy útil. Este algoritmo toma un arreglo de datos y un operador binario asociativo y devuelve un elemento que es el resultado de aplicarle el operador a los primeros dos elementos y subsecuentemente aplicarlo al resultado obtenido y al siguiente elemento, resultando al final con un solo elemento. Este algoritmo es utilizado en muchos problemas y en particular se logra una implementación muy eficiente en la arquitectura CUDA. Cabe señalar que la complejidad asintótica de este algoritmo secuencial es de $O(n)$ donde n es el tamaño del arreglo mientras que en CUDA C la complejidad asintótica es de $O(\log\ n)$.

8.1. Motivo de estudio.

Los problemas geométricos, desde el punto de vista matemático, han acompañado a la humanidad desde los tiempos antiguos. En particular, la geometría *euclidiana*[1] ha sido estudiada, formalizada y desarrollada por muchos años e incluso hoy en día, forma parte de la enseñanza básica en las matemáticas.

Si se considera que el desarrollo de algoritmos modernos se crea para resolver problemas científicos, en particular matemáticos, no es de extrañarse que haya muchos problemas

[1]Llamada así en honor al matemático de la Grecia antigua Euclides, quien describió los principios fundamentales de esta geometría y que son conocidos como "los cinco postulados de Euclides".

geométricos que requieran ser resueltos mediante una computadora. A la rama de las ciencias de la computación encargada de diseñar y estudiar algoritmos que resuelvan problemas geométricos se le conoce como *geometría computacional*. El problema de hallar la pareja de puntos más cercana en un plano es un problema geométrico y es entonces estudiado por esta rama.

Este problema se eligió porque muchos de los problemas en esta rama son resueltos por algoritmos *divide y vencerás* y más aún, algunos tienen relación entre sí como *diagramas de Voronoi* y *triangulación de Delauney* o el problema del *cierre convexo* y el problema de *ordenación*.

La elección particular de este problema fue porque ni el planteamiento ni la solución requieren conocimientos profundos de geometría sino solo nociones básicas como: calcular distancias o rectas que dividan al plano en semiplanos.

8.2. Planteamiento del problema.

Es fácil idear un algoritmo sencillo para resolver el problema: este consiste en calcular la distancia entre cualesquiera dos puntos e ir recordando la menor distancia vista hasta ahora. Este algoritmo tiene una complejidad temporal lineal ($O(n)$ con respecto al número de distancias y el problema radica ahí, pues si se calcula la distancia entre cualesquiera dos puntos el número total de distancias es de orden asintótico cuadrático, es decir hay $O(n^2)$ distancias en un conjunto de n puntos por lo que calcularlas todas llevaría $O(n^2)$ dando como resultado que el algoritmo esté en esa clase de complejidad.

Existe un algoritmo *divide y vencerás* que resuelve el problema más eficientemente. En la siguiente sección se presenta este algoritmo.

8.3. Un algoritmo secuencial.

La idea principal del algoritmo es utilizar la noción de *divide y vencerás*. Primero se divide al conjunto de puntos $P = \{p_1, p_2, \ldots p_n\}$ en el plano con una recta paralela al eje y; es decir, una recta con ecuación de la forma $x = k$. Esta recta dividirá al conjunto en dos partes. Para ello, se considera al conjunto de puntos ordenado por su coordenada x, es decir, p_1 es el punto con menor coordenada x y p_n el que tiene coordenada x más grande. Así, la recta debe pasar por el punto medio en la coordenada x entre $p_{n/2}$ y $p_{n/2+1}$.

Este algoritmo se utiliza de manera recursiva para encontrar a la pareja de puntos más cercana en cada una de las partes. Sin embargo, las parejas que no se han considerado son aquellas que tienen a un punto de un lado de la recta divisoria y a otro del otro. Ahora, esta es la parte fundamental del algoritmo pues si se acota el trabajo realizado para revisar estas parejas que tengan a sus elementos divididos por la recta a un trabajo en $O(n)$, entonces la complejidad del algoritmo es la que queremos: $O(n \log\ n)$.

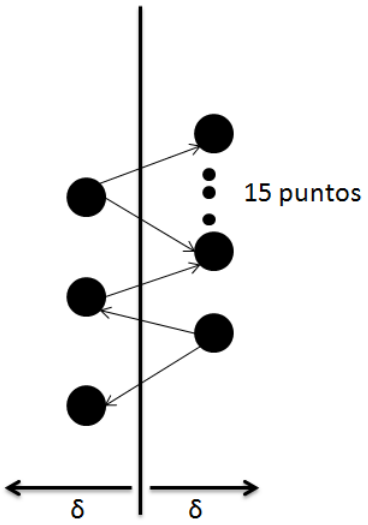

Figura 8.1: Solo se necesitan verificar un número constante de puntos por cada punto del conjunto S_y.

Dado que solo se necesita verificar un máximo de 15 parejas por punto en este conjunto, este paso toma $O(n)$, [14]. Este hecho se observa en la Figura 8.1.

El algoritmo funciona como sigue: se construyen las listas de puntos P_x y P_y que son las listas ordenadas de los puntos de entrada por sus coordenadas x e y respectivamente. Esto toma $O(n \log\ n)$ operaciones. Después, se utilizan estas dos listas como conjuntos de trabajo. Si son tres puntos o menos, el cálculo de la pareja más cercana se hace por *fuerza bruta*, calculando todas las posibles distancias. Esto funciona como el caso base de la recursión.

Después, se divide el conjunto de puntos de entrada en dos partes mediante una recta simbólica paralela al eje y. En este paso se forman cuatro listas de puntos: L_x, L_y que son los puntos a la izquierda de la recta l ordenados respectivamente por sus coordenadas x e y y R_x,R_y que son los puntos a la derecha de la recta ordenados de forma análoga, esto toma $O(n)$.

A continuación, se hace recursión con las listas L_x, L_y y R_x, R_y y se obtiene la menor de estas dos distancias. El resultado de cada una de estas llamadas recursivas es, por supuesto, la pareja de puntos más cercana en las mitades izquierda y derecha del conjunto original. A esta distancia mínima la llamamos δ.

Después se considera al punto x' que es el punto con mayor coordenada x de L_x y se construye el conjunto S que son todos los puntos a distancia menor o igual a δ de la recta l que pasa por x'. A partir de este conjunto se obtiene S_y que es el conjunto S ordenado por su coordenada y. Recordemos que como se tiene a P_y, S_y puede construirse en $O(n)$ ya que solo se filtran los puntos en P_y que cumplan la condición anterior.

Ahora, por cada punto en S_y se calcula la distancia a los 15 puntos siguientes en S_y manteniendo la mínima distancia de todas estas. Como S_y tiene tamaño lineal con respecto al tamaño de la entrada y como el proceso de cada elemento de S toma tiempo constante, ya que solo se procesan las siguientes 15 entradas en S_y, el proceso completo de los puntos en S toma $O(n)$ operaciones.

Finalmente, se compara la distancia obtenida del proceso anterior (la mínima de todas las calculadas) con δ regresando la menor de las dos obteniendo el resultado. A partir de esta descripción, se puede observar que la complejidad temporal de este algoritmo es $O(n \log\ n)$ mejorando asintóticamente el primer algoritmo descrito en la sección anterior. En el Algoritmo 8.1 se encuentra el pseudocódigo.

Algoritmo 8.1 Algoritmo secuencial y recursivo para encontrar la pareja de puntos más cercana.

Entrada P_x y P_y los puntos de $P = \{p_1, p_2, \ldots p_n\}$ ordenados por sus coordenadas x e y respectivamente.
Salida La pareja de puntos más cercana en P.
if $|P| < 3$ **then**
Hallar la pareja más cercana comparando todas las posibles distancias.
end if
Construir L_x, L_y, R_x y R_y.
$(l_0, l_1) = closestPair(L_x, L_y)$; $(r_0, r_1) = closestPair(R_x, R_y)$
$\delta = min(d(l_0, l_1), d(r_0, r_1))$
x' es el punto con mayor coordenada x en L ; $l = \{(x, y)|x = x'\}$
S es el conjunto de puntos a distancia menor o igual a δ de l ; Construir S_y.
for all $s \in S_y$ **do**
Calcular distancia (d) entre s y los siguientes 15 puntos a s en S_y
end for
(s, s') es el par que tiene menor distancia de todos.
if $d(s, s') < \delta$ **then**
return $d(s, s')$
else
return δ
end if

8.4. Estrategia de paralelización.

Para paralelizar este problema se puede pensar en adaptar el algoritmo *divide y vencerás* presentado en la sección anterior. Para lograrlo, se transforma el algoritmo a una forma iterativa de abajo hacia arriba donde cada una de estas iteraciones se paraleliza. Esto se logra paralelizando cada procedimiento en dicha etapa; es decir, el cálculo del conjunto S_y y la obtención del menor candidato de S.

Otra estrategia de paralelización es utilizar el algoritmo ingenuo y hacer que los hilos calculen las posibles distancias y mantener la menor de todas. Aunque el cálculo de las distancias se puede hacer muy fácilmente en paralelo el problema radica en encontrar la menor distancia de todas. Una primera idea es utilizar una variable compartida por todos los hilos, y cada vez que los hilos encuentren una distancia menor al valor de esta variable actualicen este valor de manera atómica. Esta estrategia resulta ineficiente debido a un problema en computación paralela y concurrente conocido como *contención*. Este problema surge cuando muchos hilos

quieren leer o escribir una variable compartida pues debido a la naturaleza de estas operaciones debe de haber sincronización entre los hilos y dichas lecturas o escrituras pueden llegar a serializarse (dependiendo de la arquitectura paralela concreta). En este caso, al utilizar una operación atómica sobre una variable en (está en ya que es compartida por todos los hilos) las escrituras se realizan de manera secuencial, causando un grave cuello de botella y haciendo más lento todo el algoritmo. Por supuesto, entre más hilos haya, el número de escrituras secuenciales aumenta y el problema se vuelve más grave.

Sin embargo, no todo está perdido con esta idea de utilizar el algoritmo ingenuo. Dado que el problema es encontrar la menor de las distancias, una primera optimización para el procedimiento descrito en el párrafo anterior es que un solo hilo calcule muchas distancias de manera secuencial y se quede con la menor de todas las que él calculó. Así, mejora la utilización de recursos, disminuye el número total de hilos necesarios para realizar el trabajo y se realizan menos operaciones atómicas pues ahora cada hilo realiza solo una.

Una mejora a lo anterior sería que solo se realizara una operación atómica por bloque. Para lograr esto primero todos los hilos del bloque deben trabajar en conjunto para obtener el mínimo del bloque. La manera en la que esto se logra es utilizando el algoritmo de *reducción* con la operación del mínimo de dos elementos que es un operador binario y asociativo. Es decir, en este caso, con el operador mínimo, reducir un arreglo significa quedarse siempre con el mínimo de los elementos de todo el arreglo por lo que si se reduce un arreglo de candidatos mínimos se obtiene el mínimo total.

Esta reducción se realiza en etapas en paralelo. Primero se reduce un arreglo aplicando el operador cada dos elementos. Después, se reduce el resultado y así sucesivamente hasta obtener solo un elemento. Con esto se obtiene un elemento por bloque. Ahora hay dos opciones, dejar que cada bloque mediante una operación atómica escriba su mínimo o realizar otra etapa de reducción para obtener el mínimo global. Con estas modificaciones este algoritmo se vuelve más viable y de hecho, se implementó de esta manera.

8.5. Algoritmo paralelo en CUDA C .

En esta sección se detalla la implementación en CUDA C de ambas estrategias: la del algoritmo ingenuo y la del adaptado a partir de la versión secuencial.

8.5.1. Algoritmo ingenuo.

Como ya se comentó en la sección anterior este algoritmo se implementa utilizando un *kernel* en donde cada hilo calcula la distancia entre dos puntos, la elección de estos puntos está dada por el identificador global del hilo; es decir, el hilo i ejecuta la distancia i en el orden lexicográfico de los puntos. Subsecuentemente por medio de un `while` calcula otras distancias siempre que su identificador actual sea menor que el número de estas distancias. En cada iteración se determina la siguiente distancia a calcular por medio de un *stride*. Así mismo, cada que se calcula una distancia se verifica si esta es menor que la más pequeña encontrada

Código 8.1: Primeras etapas de reducción de mínimos a nivel de bloque.

```
if(block_size >= 2048){
 if(tid < 2048){
  if(tid + 2048 < block_size && minimos_parciales[tid+2048] < minimo )
   minimos_parciales[tid] = minimo = minimos_parciales[tid+2048];
  }
  __syncthreads();
}
if(block_size >= 1024){
 if(tid < 1024){
  if(tid + 1024 < block_size && minimos_parciales[tid+1024] < minimo )
   minimos_parciales[tid] = minimo = minimos_parciales[tid+1024];
  }
  __syncthreads();
}
...
```

por este hilo hasta ahora (distancias calculadas en iteraciones previas) en cuyo caso desecha la distancia anterior y guarda la nueva.

Al término del `while` el hilo escribe su mínimo en un búffer compartido con los otros hilos del bloque y se sincroniza con los otros mediante una sincronización a nivel de bloque. Después, inicia la etapa de reducción dentro del bloque para hallar la mínima distancia encontrada por todo el bloque.

Esta reducción, como ya se mencionó, se hace por etapas. Después de cada etapa el conjunto de mínimos se ha reducido a la mitad. En cada etapa la primera mitad de los hilos (aquellos que tengan un identificador menor a la mitad del tamaño de la etapa actual) verifican si la distancia en su posición es menor a la distancia en la posición más el salto a la derecha (en este caso el salto es el tamaño de la etapa). La primera etapa contempla si hay el máximo número de hilos por bloque y en cada etapa este número se va reduciendo a la mitad. En el Código 8.1 se presenta un pequeño fragmento del programa que representa esta reducción por etapas; `block_size` es el tamaño de los elementos a procesar.

Como se observa a partir del código, se empieza con la etapa más grande y en cada `if` se va reduciendo el tamaño del conjunto de trabajo. El arreglo `minimos_parciales` se encuentra en memoria compartida. Además, debido a la estructura del código, si un hilo se salta una etapa; es decir, si su identificador es mayor o igual que el de la condición, se saltará las demás por lo que terminará su ejecución. Siguiendo este argumento, el hilo que ejecuta todas las etapas es el primer hilo de cada bloque.

Otro detalle de implementación es que al llegar a etapas de tamaño menor o igual a 32, la reducción la hará solo un *warp* y como las instrucciones en CUDA son ejecutadas de manera secuencial por todos los hilos de un *warp* la sincronización se hace inecesaria, por lo que el código de estas últimas etapas queda como el expuesto en el Código 8.2.

Finalmente, el primer hilo de cada bloque escribe en el búffer de salida el valor de `mínimo` en

Código 8.2: Reducción de mínimos a nivel de *warp*.

```
if(tid < 32){
 if (block_size >= 32) {
  if(tid + 32 < block_size && minimos_parciales[tid+32] < minimo )
    minimos_parciales[tid] = minimo = minimos_parciales[tid+32];
 }
 if (block_size >= 16) {
   ...
 }
 if (block_size >= 8) {
   ...
 }
 .
 .
 .
 if (block_size >= 1) {
  if(tid + 1 < block_size && minimos_parciales[tid+1] < minimo )
    minimos_parciales[tid] = minimo = minimos_parciales[tid+1];
 }
}
```

la posición correspondiente de acuerdo al número de bloque al que pertenece.

Hasta este punto en el búffer de salida se tiene el mínimo de cada bloque. Si solo hay un bloque, el programa termina aquí y muestra el resultado; de lo contrario, se ejecuta otro *kernel* muy similar donde solo se hacen las etapas de reducción ahora tomando como búffer de entrada el búffer de salida del *kernel* anterior. Después de este segundo *kernel* se obtiene el resultado final pues este se lanza con solo un bloque.

8.5.2. Algoritmo *divide y vencerás.*

La implementación se hizo adaptando la versión secuencial paso a paso. La idea es utilizar la misma estrategia de partición de puntos de abajo hacia arriba, así que el primer paso es resolver los casos base, que en este caso son los conjuntos de tres puntos. Para esto se lanza un *kernel* que calcule la mínima distancia de cada uno de estos conjuntos y la anote en un búffer de salida.

Después, se inicia una iteración de etapas en donde en cada etapa primero se lanza un *kernel* para inicializar datos en los buffers de salida, después se utiliza otro para calcular el conjunto S de cada uno de los bloques de trabajo de la etapa correspondiente, por ejemplo, en la primera etapa se calcula el conjunto S correspondiente a cada uno de los conjuntos de seis puntos que se procesarán en dicha etapa.

Para encontrar el conjunto S se filtran en paralelo los elementos que quedan a distancia menor o igual que δ de la línea divisoria del bloque. Después, estos elementos se encolan por

medio de operaciones atómicas. Para encontrar el conjunto S_y de cada bloque, se ordenan por y los elementos de cada bloque S encontrados usando un *merge*.

Después, se lanza otro *kernel* para calcular el mínimo de cada conjunto S_y, en este se verifica cada punto con sus siguientes 15, tal como en el caso del algoritmo secuencial. Finalmente, se lanza un *kernel* más para actualizar los resultados y calcular el mínimo del conjunto de trabajo de la etapa actual.

La implementación es muy parecida a la secuencial y utiliza muchos más *kernels* por etapa que los casos de estudio anteriores. Esto se debe a que no se ideó otra manera de adecuar el algoritmo debido a la naturaleza geométrica del problema (los otros casos de estudio tenían naturaleza aritmética).

Como en cada etapa se lanzan varios *kernels*, el costo administrativo de este algoritmo es alto; pues para que un *kernel* se ejecute, se deben realizar varias operaciones administrativas. En este caso, el gran número de *kernels* hace que estas operaciones tengan un impacto negativo en la eficiencia del programa.

A pesar de que esta implementación es más complicada y de que utiliza muchos más *kernels* y datos administrativos que la implementación ingenua, tiene una mejor complejidad asíntótica con respecto a la ingenua. Lo anterior se ve reflejado en los resultados obtenidos.

8.6. Resultados y conclusiones.

Tras haber hecho las tres implementaciones y haberlas ejecutado en la computadora de pruebas (la descrita en la sección 5.6) se consiguieron resultados interesantes.

La versión ingenua es más eficiente que la versión secuencial en conjuntos de puntos alrededor de diez mil. Sin embargo, debido a la sobrecarga de trabajo que se requiere para calcular los puntos a evaluar o la memoria necesaria para almacenar este cálculo antes de iniciar la ejecución, esta implementación deja de ser eficiente y viable para conjuntos de puntos más grandes. Esto sin contar que una de las razones para implementar algo en CUDA C es tener tiempos de ejecución aceptables con conjuntos de datos muy grandes, por lo que esta versión queda completamente descartada.

Por otro lado, para la versión paralela *divide y vencerás* se observa que su rendimiento sí es mejor que el de la versión secuencial, en la Figura 8.2 se muestra la comparación de rendimiento entre las versiones *divide y vencerás* en CUDA C y secuencial.

En la Figura 8.3 observa que la mejora de la versión en CUDA C es menor con respecto a los otros casos de estudio. Esto es un resultadoe esperado, pues la implementación en CUDA C es muy similar a la implementación secuencial, es decir, no se hicieron grandes mejoras algorítmicas como en el caso de *mergesort* ni se explotó de sobremanera el paralelismo subyacente como en la transformada rápida de Fourier.

A pesar de que la tarjeta utilizada tiene capacidad de cómputo 3.5, los programas se compilaron utilizando la capacidad 2.0. Esto porque después de haber hecho pruebas, al compilar

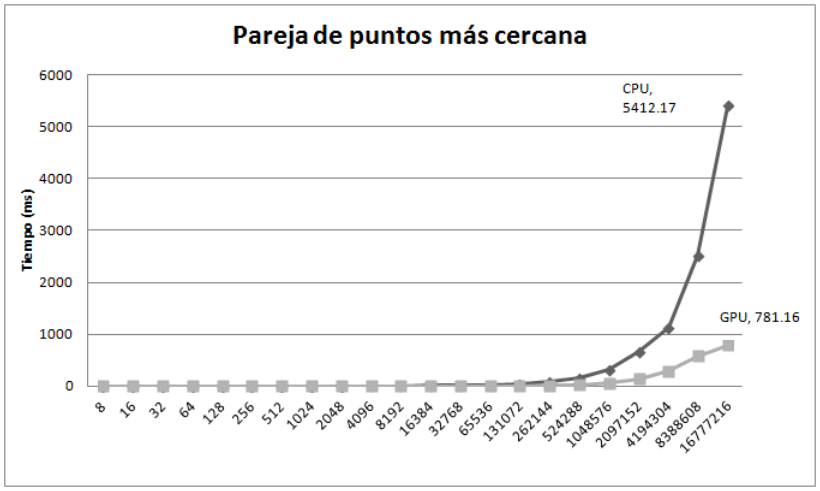

Figura 8.2: Comparativa de rendimiento del algoritmo de la pareja de puntos más cercana

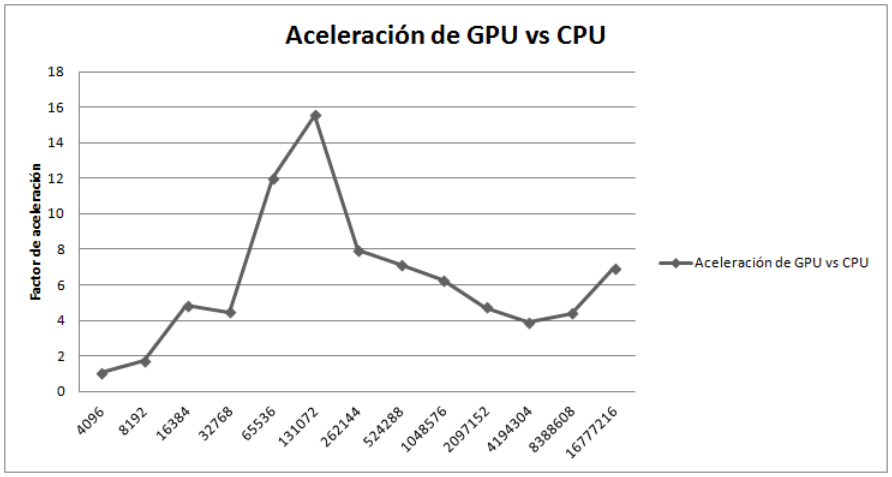

Figura 8.3: Gráfica de aceleración del algoritmo de la pareja de puntos más cercana

con esta versión, se obtenía el menor tiempo de ejecución comparado con compilar con otras capacidades, en particular con la 3.5 y la 3.0 resultando esta última la menos eficiente.

Conclusiones generales y trabajo futuro.

Al analizar los resultados de los casos de estudio se puede notar que no todos los algoritmos *divide y vencerás* son buenos candidatos para ser paralelizados en CUDA. A pesar de que se obtuvieron mejoras en los tres casos de estudio, en los casos donde el problema a resolver era de naturaleza aritmética se pudo idear un algoritmo paralelo un tanto diferente al secuencial, esto muestra que la estrategia a usar en cómputo paralelo varía de acuerdo al problema aunque tengan el mismo paradigma, a diferencia del cómputo secuencial.

Los criterios para medir el rendimiento en cómputo secuencial usualmente son el tiempo y la memoria usada, aunque por lo general se prefiere al primero como criterio para decidir si un algoritmo es eficiente o no. En este tipo de arquitecturas se busca implementar un algoritmo que tenga una cota asintótica óptima y luego tratar de optimizarlo un poco más de acuerdo al lenguaje de programación empleado.

En cómputo paralelo, en cambio, hay varios criterios para medir la eficiencia de un programa. Además del tiempo y la memoria usados se puede pensar en el número de procesadores necesarios o en el número de operaciones que se deben realizar para resolver el problema.

A fin de cuentas, se trató de que se maximizará siempre la ocupación de la tarjeta, para esto se utilizó una calculadora de ocupación provista por NVIDIA cuando se instala CUDA. Con esto, los factores del número de hilos y el trabajo que realizan pasan a segundo plano y el enfoque fue utilizar el tiempo como principal factor de rendimiento y en particular en comparar el tiempo utilizado por las versiones secuenciales contra el tiempo de las versiones paralelas en CUDA C.

El último caso de estudio es el único en donde el algoritmo paralelo sigue la misma idea que el secuencial. La adaptación simplemente es tratar de paralelizar cada paso requerido en una etapa del árbol de recursión el algoritmo secuencial. Esto lleva a que en cada etapa haya varios lanzamientos de *kernels* a diferencia de los otros problemas en donde había solo uno o dos lanzamientos por etapa. A pesar de esto, se logra una mejora con respecto al algoritmo secuencial. Esto ocurre, sobretodo, cuando el tamaño de la entrada aumenta debido a que en los primeros niveles del árbol hay muchos nodos que pueden ser resueltos de manera paralela. A pesar de esto, el paralelismo se pierde conforme se va subiendo en el árbol y en los últimos niveles se utilizan pocos hilos para completarlos.

En los primeros casos también se lograron mejoras significativas, hablando del caso de la transformada rápida de Fourier, se logra un algoritmo en donde no se pierde paralelismo

conforme se ejecutan etapas en el árbol de recursión. Esto permite que el dispositivo esté ocupado en la misma medida durante todo el algoritmo. Cabe mencionar que una implementación ingenua de este algoritmo podría ocasionar que el paralelismo se pierda. En particular el paralelismo se perdería por un factor de $\frac{1}{2}$ en cada etapa del árbol de recursión, llegando al extremo de que en la última etapa solo se podría utilizar un hilo y se haría de forma secuencial.

Un posible trabajo futuro con respecto a este problema podría ser realizar utilizando sincronización de bloques como se hizo en el caso de *merge*. También se pueden buscar otras aplicaciones para la FFT y realizar una aplicación real que utilice el código aquí presentado.

En cuanto a *merge* y *mergesort* los resultados más relevantes son en el primer caso de estudio. En el trabajó se ideó una manera de lograr paralelismo dinámico. Aunque, a la fecha las nuevas versiones de CUDA ya lo traen de manera nativa, no consiste en lo que se necesita, pues CUDA solo permite lanzamiento anidado de *kernels*. Por otro lado, se encontró un algoritmo masivamente paralelo en donde se usa la idea original de la búsqueda binaria pero de una manera que se adapta de gran manera a la arquitectura paralela. Se debe mencionar que el haber encontrado ese algoritmo y haberlo implementado causó una gran satisfacción pues se aprendió a pensar de mejor manera como adaptar los algoritmos secuenciales en CUDA C para que se ajusten a la arquitectura concreta.

Como trabajo futuro para estos problemas, se podría tratar de mejorar aún más el algoritmo utilizando la técnica mostrada en [23].

En cuanto al problema de la pareja de puntos más cercana, aunque la implementación ingenua en CUDA C no resultó muy eficiente para un número grande de datos, la idea del algoritmo de *fold* mediante memoria compartida y sincronización por bloque es muy útil y muestra el potencial que tiene el cómputo masivamente paralelo en las GPU para algunos problemas.

Finalmente, se concluye que hay algoritmos que se paralelizan y adaptan fácilmente a CUDA C y que logran una gran mejora contra sus partes secuenciales. Otros que en principio no exhiben paralelismo (como *merge*) pero que con otra estrategia el paralelismo logrado es muy satisfactorio para el programador y adaptable a CUDA C. Hay otros que cuando se paralelizan, el algoritmo no se ajusta eficientemente a la arquitectura y es poco conveniente invertir tiempo y esfuerzo para implementarlo. Sin embargo, como en el caso de *mergesort* al paralelizarse eficientemente una parte del algoritmo, el algoritmo completo puede mejorar drásticamente.

Así mismo, de acuerdo con la investigación de artículos y aplicaciones, así como por experiencia propia, para realizar implementaciones paralelas en las GPU se debe estudiar no solo el problema a resolver; sino también, la arquitectura misma y el lenguaje de programación subyacente. Pues aunque todos los casos de estudio tienen el paradigma *divide y vencerás* en cómputo secuencial, las implementaciones de los mismos en CUDA C no son iguales, y de hecho difieren tanto en su estructura como en su nivel de paralelismo. Todo lo anterior lleva a no darse por vencido en el trayecto de paralelizar y mejorar algoritmos ya óptimos y eficientes mediante las GPU, pues hoy en día las mejores computadoras del mundo están basadas en el cómputo en GPU y cuentan con decenas o centenas de miles de núcleos. Y no

solo las súper computadoras cuentan con estas características, ya hoy en día los dispositivos móviles o las consolas de videojuegos también cuentan con GPU con muchos núcleos.

Para terminar, este trabajo sirvió para aprendizaje no solo de algoritmos, en particular de algoritmos *divide y vencerás*, sino también como base para aprender cómputo paralelo, pues aunque la implementación de los algoritmos depende totalmente de una arquitectura masivamente paralela, cada vez el número de procesadores se incrementa tanto en las computadoras personales como en las GPU o en los coprocesadores. No sería de extrañarse que en unos años se puedan aplicar estas ideas masivamente paralelas en los procesadores que traigan las computadoras portátiles o los dispositivos móviles. Lo cierto es que el futuro es incierto y casi siempre llega antes de lo que se esperaría.

Apéndice A

Glosario de términos y siglas.

Anfitrión o *host*. Es el nombre con el que se conoce a la computadora (CPU) que ejecuta las instrucciones necesarias administrar la GPU. En CUDA C las funciones ejecutadas por el dispositivo son invocadas desde el anfitrión.

Archivo de registros. Conjunto de registros asociados a cada SM. Las variables locales de los SP son asignadas a un registro en este archivo.

Bloque. Unidad administrativa que engloba a un grupo de hilos. Un bloque es asignado a un SM y el SM calendariza al bloque para que los hilos del bloque ejecuten instrucciones. Todo bloque debe tener al menos un hilo y un *kernel* debe ser invocada con al menos un bloque.

Caché L1. Tipo de memoria caché presente en las GPU actuales. Cada SM tiene su caché L1 y estos son independientes entre sí. El tamaño actual es de 64 KB por SM y esta capacidad es compartida por la memoria compartida.

Caché L2. Tipo de memoria caché presente en las GPU actuales. Este caché es compartido por todos los SM y alcanza hasta los 768 KB.

Contención. Fenómeno que ocurre cuando varios hilos están tratando de acceder a un recurso compartido. Este fenómeno lleva a un pobre rendimiento y siempre se debe tratar de eliminar o en su caso, minimizar.

Digráfica de dependencias. Representación matemática de las dependencias entre los subproblemas recursivos en un algoritmo *divide y vencerás*.

Divergencia de hilos. Fenómeno que ocurre cuando hilos del mismo *warp* ejecutan instrucciones diferentes. Los hilos involucrados ejecutan las instrucciones de manera secuencial y esto ocasiona un pobre rendimiento.

GDDR: *Graphics Double Data Rate*. Tipo de memoria RAM presente en las GPU. La memoria RAM presente en CUDA es GDDR5.

Hilo. Unidad administrativa que se encarga de ejecutar instrucciones en CUDA. Un hilo en CUDA C es asignado a un SP en un SM. El SP ejecuta las instrucciones del hilo.

Interbloqueo. Problema de sincronización que ocurre cuando ningún hilo puede continuar con su ejecución ya que espera a que otro hilo realice ciertas operaciones. En este problema ningún hilo puede continuar ejecutando sus intrucciones por lo que el programa en su totalidad se bloquea.

Kernel. Función principal de dispositivo que se ejecuta desde un anfitrión. Al momento de la invocación se debe especificar el número de bloques y el número de hilos por bloque que ejecutarán esta función.

Latencia de memoria. Es el tiempo que tarda una operación de acceso a memoria en ser completada.

Línea de caché. Tamaño mínimo de información que se almacena en el caché. Si una parte de la línea deja de ser válida, la línea completa se desecha y se vuelve a obtener de la memoria global o de otro nivel de caché según corresponda. El tamaño de una línea de caché en CUDA es de 128 bytes.

Malla. Conjunto de bloques lanzados al invocar un *kernel*. Una malla se compone por varios bloques, que a su vez, se componen por varios hilos.

Memoria compartida. Tipo de memoria específico en una GPU. Este tipo de memoria es parte del caché L1 a partir de la segunda generación de GPU de NVIDIA. Esta memoria está presente en cada SM y es limitada. Al compartir espacio con el caché L1 es de acceso muy rápido y compartida por todos los hilos de un bloque.

Memoria global. Tipo de memoria principal y más abundante en una GPU. Las GPU actuales tienen hasta 4 GB de memoria global.

Ocupación del dispositivo. Es la medida que determina cuánto se están utilizando los recursos disponibles en el dispositivo.

Operación atómica. Tipo de operación implementada en hardware o software que asegura que o se ejecuta sin interrupciones de hardware o software o no se ejecuta en sí. Se dice que son atómicas porque son "indivisibles" desde el punto de vista de los hilos en ejecución.

Paralelismo dinámico. Estrategia de implementación de paralelismo donde se permite crear hilos y fusionar resultados de manera en tiempo de ejecución mediante instrucciones o funciones.

Paralelismo precalculable. Tipo de paralelismo en donde en tiempo de compilación se puede conocer cómo serán las tareas recursivas de un problema *divide y vencerás*.

Pérdida de paralelismo. Fenómeno que ocurre en los problemas paralelos resueltos por etapas. En cada etapa el número de hilos necesarios para resolverla disminuye. Este fenómeno usualmente ocasiona un pobre rendimiento paralelo.

***Pipeline* gráfico.** Ancestro de lo que hoy se conoce como GPU. Los *pipeline* gráficos evolucionaron para trasformarse en una GPU moderna.

Sección crítica. Porción del código que puede ser accedida por solo un hilo de ejecución a la vez. Si más de un hilo ejecuta las instrucciones de esta sección a la vez, puede haber

problemas de consistencia de datos.

Sincronización de barrera. Primitiva de sincronización donde se establece un punto de encuentro entre los hilos en ejecución. Ningún hilo continúa su ejecución hasta que todos hayan alcanzado y ejecutado la instrucción de barrera.

SP: *Stream Proccesor* o núcleo CUDA. Es una unidad de hardware presente en cada SM. Esta unidad se encarga de ejecutar las instrucciones de los programas en paralelo junto con los otros SP del mismo SM.

SM: *Stream Multiproccesor*. Uno de los componentes principales de una GPU junto con la memoria global. Una GPU tiene al menos un SM y puede llegar a tener hasta 16. En un SM residen decenas de núcleos, además de otros componentes que permiten la ejecución de programas generales.

***Stream*.** En el ámbito de CUDA C, un *stream* es la manera de invocar *kernels* a modo de *pipeline*, es decir, kernels de un mismo *stream* se invocan de manera sucesiva uno detrás del otro, pero *kernels* de distintos *streams* pueden ejecutarse a modo de *pipeline*.

***Warp*.** Unidad de asignación de hilos a SP. Un SM asigna y calendariza *warps* en sus núcleos. Todo hilo pertenece a un *warp* e hilos del mismo *warp* ejecutan sus instrucciones de manera síncrona.

Bibliografía

[1] *Intel Pentium 4 3.06GHz CPU with Hyper-Threading Technology: Killing Two Birds with a Stone...*, Noviembre 2002. `http://www.xbitlabs.com/articles/cpu/display/pentium4-3066.html#sect0`, visitado por última vez: 15/ago/2014.

[2] *CUDA C Best Practices Guide*, Febrero 2014. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`.

[3] *CUDA C Programming Guide*, Febrero 2014. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[4] *Dependency Flow Graph Example*, 2014. `https://software.intel.com/en-us/node/506216`.

[5] Alcántara, Manuel: *Programación dinámica paralela en las GPU.* Tesis de maestría, Universidad Nacional Autónoma de México., 2014.

[6] Atallah, Makhail J.: *Efficient Parallel Solutions to Some Geometric Problems*, 1985.

[7] Batcher, K. E.: *Sorting Neyworks and their Applications.* 1968.

[8] Cook, Shane: *CUDA Programming. A Developer's Guide to Parallel Computing with GPUs.* Morgan Kaufmann, primera edición, 2013.

[9] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest y Clifford Stein: *Introduction to Algorithms.* The MIT Press, tercera edición, 2009.

[10] Garland, Michael: *Algorithm Design for Manycores GPUs*, 2009.

[11] Herlihy, Maurice y Nir Shavit: *The art of multiprocessor programming.* Morgan Kaufman, primera edición, 2008.

[12] Kernighan, Brian W. y Dennis M. Ritchie: *The C Programming Language.* Prentice Hall, segunda edición, 1988.

[13] Kirk, David B. y Wen Mei W. Hwu: *Programming Massively Parallel Proccesors.* Morgan Kaufmann, segunda edición, 2013.

[14] Kleinberg, Jon y Éva Tardos: *Algorithm Design.* Addison Wesley, primera edición, 2006.

[15] Ladner, Richard E. y Michael J. Fischer: *Parallel Prefix Computation.* 1980.

[16] Merrill, Duane y Andrew Grimshaw: *Parallel Scan for Stream Architectures*, 2009.

[17] NVIDIA: *Whitepaper. NVIDIA GF100. World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism*, 2010.

[18] NVIDIA: *Whitepaper. NVIDIA GeForce GTX 680. The fastest, most efficient GPU ever built*, 2012.

[19] NVIDIA: *Whitepaper. NVIDIA's Next Generation CUDA Compute Architecture*, 2012.

[20] Reinhard, Diestel: *Graph Theory.* Springer, tercera edición, 2005.

[21] Rubio, M., P. Gómez y Drouiche: *A new superfast bitreversal algorithm.* 2001.

[22] Sanders, Jason y Kandrot Edward: *CUDA by example. An Introduction to General-Purpose GPU Programming.* Addison Wesley, primera edición, 2010.

[23] Satish, Nadathur, Mark Harris y Michael Garland: *Designing Efficient Sorting Algorithms for Manycore GPUs.* 2009.

[24] Sengupta, Shubhabrata, Mark Harris y Michael Garland: *Efficient Parallel Scan Algorithms for GPUs*, 2008.

[25] Sengupta, Shubhabrata, Mark Harris, Michael Garland y John D. Owens: *Efficient Parallel Scan Algorithms for Many-core GPUs*, 2008.