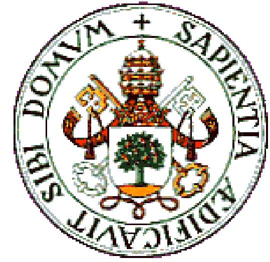
UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN



TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

PARALELIZACIÓN DE ALGORITMOS PARA EJECUCIÓN EN ENTORNO MULTI-GPU

AUTOR: ALBERTO GALLEGO LUENGO

TUTOR: MARIO MARTÍNEZ ZARZUELA

14 de septiembre de 2015

A mi padre y a mi madre,
por ofrecerme una vida tan fácil.

TÍTULO:	**Paralelización de algoritmos para ejecución en entorno multi-GPU**
AUTOR:	**Alberto Gallego Luengo**
TUTOR:	**Mario Martínez Zarzuela**
DEPARTAMENTO:	**Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE:	**Míriam Antón Rodríguez**
VOCAL:	**David González Ortega**
SECRETARIO:	**Mario Martínez Zarzuela**
SUPLENTE:	**Francisco Javier Díaz Pernas**
SUPLENTE:	**Carlos Gómez Peña**

FECHA:	**14 de septiembre de 2015**
CALIFICACIÓN:	

Resumen del TFG

El objetivo principal de este Trabajo Fin de Grado es el desarrollo de un código de programa reutilizable para aplicaciones científicas que haga uso de la programación GPGPU para acelerar los cálculos y que pueda ser llamado desde MATLAB. También se realiza un estudio y comparativa de la ejecución de algoritmos en la GPU frente a la CPU, a la vez que se analizan los beneficios del empleo de streams y multi-GPU.

En primer lugar se realiza una presentación de la programación GPGPU y se explican las características más importantes del leguaje CUDA C. A continuación se comentan brevemente las funciones del programa desarrollado, para más tarde realizar una serie de prueba y presentar los resultados. Para terminar se muestran una serie de conclusiones obtenidas y se sugieren algunas posibles líneas futuras de trabajo.

Palabras clave: GPGPU (General Purpouse on Graphical Processing Units), programación paralela, CUDA C, MATLAB, archivos c-mex.

Abstract

The primary aim of this bachelor's thesis is to develop a program code, using GPGPU programming to speed up calculations, that could be reusable for scientific applications and that could be called from MATLAB. Also, a study and a comparative test of the execution of the algorithm on the GPU versus the CPU is performed, while simultaneously analyzing the benefits of using streams and multi-GPU.

Firstly, GPGPU programming is introduced and the main characteristics of CUDA C language are explained. Then, the developed program functions are shortly discussed, after that a test is made and the results are submitted. At the end a number of conclusions are outlined, and future work lines are suggested.

Keywords: GPGPU (General Purpouse on Graphical Processing Units), parallel programing, CUDA C, MATLAB, c-mex files.

Índice de contenidos

Resumen .... **IX**

Capítulo 1. Introducción **1**

1.1. Motivación y objetivos 3

1.2. Fases y métodos 4

1.3. Medios disponibles 4

1.3.1. Hardware: 5

1.3.2. Software: 5

1.4. Organización del Trabajo Fin de Grado 6

Capítulo 2. Programación de propósito general en GPU a través de NVIDIA CUDA C **8**

2.1. Introducción 8

2.1.1. Arquitecturas Hardware: CPU/GPU 8

2.1.2. Sistemas de procesamiento secuencial y paralelo 10

2.1.3. Modelos de computación 13

2.1.4. Evolución de la graphics pipeline en GPU 14

2.2. Computación de propósito general en GPU 16

2.2.1. APIs gráficos y lenguajes de programación 17

2.3. NVIDIA CUDA 18

2.3.1. Introducción 18

2.3.1.1. Uso de la GPU para computación de datos en paralelo 18

2.3.1.2. Arquitectura NVIDIA CUDA 21

2.3.2 Modelo de programación 22

2.3.2.1. Modelo de programación escalable 22

2.3.2.2. La GPU como coprocesador multi-hilo de la CPU 24

2.3.2.3. Agrupación de hilos en la GPU 24

2.3.2.4. Modelo de memoria en GPU 26

2.3.2.5. Host and Device 27

2.3.3. Implementación hardware de la GPU 28
2.3.3.1. Arquitectura SIMT: implementación hardware 29
2.3.3.3. Capacidad de cómputo 30
2.3.4. API 30
2.3.4.1. CUDA C como extensión del lenguaje C 30
2.3.4.2. Extensiones del lenguaje CUDA C 31
2.3.4.3 Tipos de vectores propios de CUDA C 32
2.3.5. Optimización del código 33
2.3.5.1. Ocupancia 33
2.3.5.2. Transferencias de datos entre host y device 33
2.3.6. Operaciones en paralelo 33
2.3.6.1. Sincronización global 34
2.3.6.2. Accesos coalescentes a memoria global del device 35
2.3.6.3. Conflictos de bancos de memoria compartida 36
2.3.7. CUDA Streams 37
2.3.7.1. El stream por defecto (default stream) 37
2.3.7.2. Streams (Non-default streams) 38
2.3.8. Multi-GPU 39
2.4. CUDA en MATLAB 41
2.4.1. Archivos c-mex 41
2.5. Ejemplos de aprendizaje 42
2.5.1. Ejemplo 1: Introducción a CUDA C 42
2.5.2. Ejemplo 2: Reservas de memoria 43
2.5.3. Ejemplo 3: Pasando argumentos al kernel y ejecución de múltiples hilos 44
2.5.4. Ejemplo 4: Eventos de CUDA 45
2.5.5. Ejemplo 5: Memoria compartida 46
2.5.6. Ejemplo 6: Streams 47
2.5.7. Ejemplo 7: multi-GPU 48

Capítulo 3. Descripción del código 50
3.1. Introducción 50

3.2. Referencia de funciones ... 52

Capítulo 4. Pruebas y resultados ... 57

4.1. Metodología de las pruebas ... 57

4.2. Comparación CPU frente a una única GPU sin usar streams ... 58

4.3. Comparación diferentes streams con una GPU ... 60

4.4. Comparación una única GPU con multi-GPU ... 63

4.5. Conclusiones ... 66

4.6. Líneas futuras ... 67

Bibliografía ... 68

Índice de figuras

Figura 1.1: Número de operaciones en punto flotante por segundo para la CPU y la GPU .2

Figura 2.1: Esquema genérico del hardware de un PC....9

Figura 2.2: Sistema SISD (Single Instruction Single Data) 11

Figura 2.3: Sistema SIMD (Single Instruction Multiple Data) 11

Figura 2.4: Sistema MISD (Multiple Instruction Single Data) 12

Figura 2.5: Sistemas MIMD (Multiple Instruction Multiple Data)....13

Figura 2.6: Modelo Von Neumann 13

Figura 2.7: Modelo Streaming....14

Figura 2.8: Arquitectura de la serie GeForce 7 de NVIDIA 15

Figura 2.9: Arquitectura de la serie GeForce 8 de NVIDIA 16

Figura 2.9: Número de operaciones en punto flotante por segundo para la CPU y la GPU19

Figura 2.10: Ancho de banda en memoria para la CPU y la GPU 19

Figura 2.11: Espacio del chip dedicado a memoria y cálculo en la CPU y la GPU....20

Figura 2.12: Pila software de CUDA....22

Figura 2.13: Escalado automático....23

Figura 2.14: Jerarquía de hilos dentro de la GPU. Una malla de bloques de hilos....25

Figura 2.15: Jerarquía de memoria en la GPU....26

Figura 2.16: Modelo de programación heterogénea, ejecución en CPU y en GPU 27

Figura 2.17: Modelo hardware de la GPU....28

Figura 2.18: Esquema de una reducción....34

Figura 2.19: Lectura/escritura coalescente con todos los hilos 35

Figura 2.20: Lectura/escritura coalescente con todos los hilos 35

Figura 2.21: Accesos a memoria permutados por los hilos 35

Figura 2.22: Dirección de inicio no alineada con la lectura/escritura de los hilos 35

Figura 2.23: Reducción con conflictos de bancos....36

Figura 2.24: Reducción sin conflictos de bancos 37

Figura 2.25: Diferentes planificaciones de streams para las mismas operaciones 39

Figura 2.26: GPUs en varias ranuras (slots) 40
Figura 2.27: varias GPUs en una única tarjeta 40
Figura 2.28: Código ejemplo "Hola Mundo" 42
Figura 2.29: Código ejemplo reserva de memoria 43
Figura 2.30: Código ejemplo eventos 45
Figura 2.31: Código ejemplo memoria compartida 47
Figura 3.1: División de los vectores en chunks y en streams 51
Figura 3.2: Esquema de las funciones del programa 53
Figura 4.1: Implementación del algoritmo en MATLAB 58
Figura 4.2: Tiempos de ejecución frente a diferentes tamaños de los vectores 59
Figura 4.3: Zoom número pequeño de elementos 60
Figura 4.4: Tiempos de ejecución frente a diferentes número de elementos y streams 61
Figura 4.5: Pocos datos y empleo de streams 62
Figura 4.6: CPU vs GPU diferentes streams para vectores grandes 62
Figura 4.7: Zoom CPU frente GPU diferentes streams 63
Figura 4.8: Diferentes streams una única GPU y multi-GPU 64
Figura 4.9: Zoom una GPU y multi-GPU utilizando 1, 2, 4 y streams respectivamente 65

Capítulo 1

Introducción

Durante más de 20 años los microprocesadores basados en una única unidad central de proceso (CPU) permitieron un rápido incremento del rendimiento computacional y una reducción del precio de los dispositivos informáticos. Esta tendencia, sin embargo, desde el año 2003 aproximadamente se ha visto frenada debido a restricciones en el consumo de energía y la disipación de calor, limitaciones que han hecho imposible continuar con el incremento de la frecuencia de reloj y el nivel de actividades productivas que se podían realizar en cada ciclo de reloj en una única CPU. Esto condujo a que los fabricantes de microprocesadores cambiaran a un modelo de CPU diferente, donde se utilizasen múltiples unidades de procesado, llamados núcleos, para poder seguir incrementando la potencia de procesado (Kirk y Hwu, 2010).

Con este cambio de paradigma, las aplicaciones software que continuaron disfrutando de un incremento del rendimiento fueron aquellas en las que múltiples hilos (*threads*) de ejecución colaboraban para completar el trabajo más rápidamente. Esta práctica de programación en paralelo no fue algo nuevo, fuera del mundo de los ordenadores de consumo doméstico, las supercomputadores llevaban décadas aprovechando las mejoras de rendimiento a través de la paralelización (Kirk y Hwu, 2010).

A la par que estos avances tenían lugar en la CPU, a partir de principios de los años 90, comenzaron a utilizarse GPUs (*Graphical Processing Unit*) que se encargaban del procesamiento de gráficos, estos cada vez más exigentes debido al auge de los videojuegos, y así aligerar la carga de la parte de la CPU (Sanders y Kandrot, 2010). A finales de esta década el número de transistores de las GPUs superó al número de transistores de las CPUs, y las diferencias entre las arquitecturas de ambos dispositivos fueron haciéndose cada vez más evidentes: la mayor parte del área del chip en las CPUs estaba dedicada a cache, mientras que en las GPUs estaba dedicada a la unidad lógica, con pequeñas caches pensadas para la agregación de ancho de banda más que para la reducción de latencias. De esta manera las aplicaciones gráficas estaban pensadas de manera tal que explotaran el paralelismo inherente entre pixeles independientes (Wilt, 2013). Para las GPUs mejorar el rendimiento incrementando el número de núcleos de ejecución, en vez de la capacidad de proceso de estos, fue una progresión natural, y desde el año 2003 han liderado la carrera en el cálculo de operaciones de precisión simple y doble (Figura 1.1) (Kirk y Hwu, 2010).

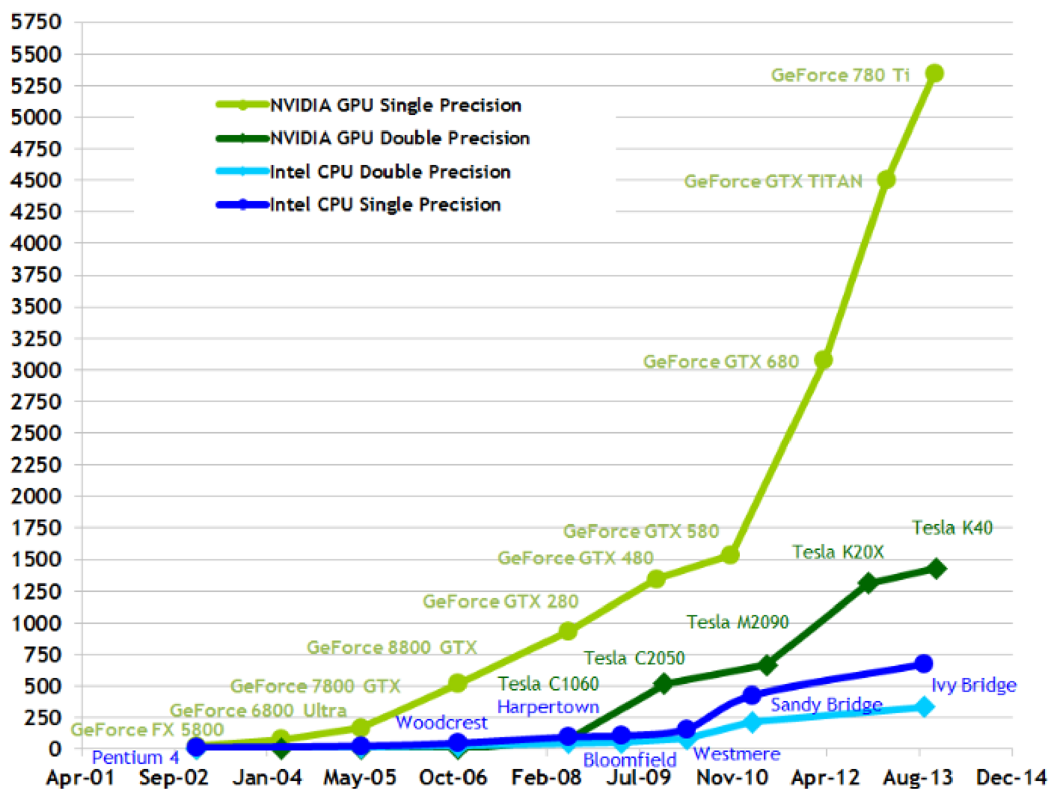


Figura 1.1: Número de operaciones en punto flotante por segundo para la CPU y la GPU

Todas estas ventajas que mostraban las GPUs, hicieron que se quisiera utilizar ese potencial no solo para tareas relacionadas con los gráficos, lo que dio lugar a lo que se conoce como GPGPU o computación de propósito general en la GPU (*General Purpose Computing on the GPU*). Hasta 2006 la única forma de comunicarse con la GPU era a través de APIs gráficas como Direct3D y OpenGL con las dificultades y limitaciones que ello conllevaba, ya que había que hacer parecer los cálculos deseados como problemas de renderizado, que era con lo que trabajaban estas APIs. Pese a todo, los excelentes resultados obtenidos mediante esta tecnología hicieron que a finales del año 2006 NVIDIA lanzara CUDA, una herramienta que permitía a los programadores en C escribir código paralelo para su ejecución en GPU usando unas pocas extensiones de lenguaje fáciles de utilizar (Wilt, 2013).

Aunque las mejoras en el rendimiento que se obtienen mediante la paralelización de los algoritmos es evidente, no todas las aplicaciones pueden ser paralelizadas, lo que nos lleva a una arquitectura de tipo heterogénea, combinando CPU y GPU. En esta situación, la CPU se encarga de las partes de código que requieran una ejecución secuencial, mientras que la GPU ejecutará aquellas partes de código paralelizable (Kirk y Hwu, 2010). Para lograr el

máximo rendimiento en estas arquitecturas heterogéneas se deben cumplir una serie de requisitos (NVIDIA Corp., 2015a):

- Minimizar el número de transferencias CPU/GPU
- Maximizar el paralelismo
- Poseer elevada intensidad aritmética
- Aprovechar anchos de banda y memoria de la GPU

1.1. Motivación y objetivos

El principal objetivo de este trabajo fin de grado es desarrollar un código reutilizable para aplicaciones científicas que haga uso de la programación GPGPU para acelerar los cálculos y que pueda ser llamado desde MATLAB. Otro objetivo es también realizar un análisis de las mejoras obtenidas por el empleo de GPGPU en comparación con el uso de la CPU. También comparamos la mejora que aporta el empleo de varias GPUs trabajando en paralelo, o lo que se conoce con el nombre de multi-GPU, así como el uso de streams.

Para ello partimos de un esqueleto de código pensado en un principio para el trabajo con imágenes, que debido al alto grado de abstracción con el que estaba escrito ha sido posible reutilizar. La operación que realiza el nuevo algoritmo es el producto escalar de dos vectores y un cálculo un tanto trivial para aumentar la carga computacional del mismo, que consiste en calcular una potencia y después hacer la raíz cuadrada del mismo para todos los elementos de un vector y repetirlo tantas veces como elementos tengan los vectores.

A la par que trabajamos en nuestro algoritmo, también respetamos y mejoramos el esqueleto de código del que partimos, separando en distintos bloques lógicos las tareas del programa. De esta manera proporcionamos una herramienta futura que sirva para migrar diferentes aplicaciones a una arquitectura multi-GPU, sin necesidad de escribir todo el código de nuevo partiendo de cero.

Una de las principales motivaciones de este trabajo reside en la mejora de las limitaciones que presenta la herramienta de cálculo en paralelo de MATLAB (*Mathwork's Parallel Computing Toolbox*) como solución para acelerar los cálculos, así como el ahorro del coste de la licencia de la misma (W. Suh y Kim, 2014).

Este trabajo se enmarca en una de las tres líneas principales de investigación del Grupo de Telemática e Imagen (GTI) de la Universidad de Valladolid: cálculos de propósito general en GPUs (GPGPUs). En este grupo se utiliza la GPGPU para el procesado de datos de redes neuronales artificiales, reconocimiento de texturas de color o movimiento, procesado de imágenes o la aceleración de análisis de datos cerebrales entre otros (Martínez Zarzuela, 2013). Gracias al trabajo del grupo GTI (Departamento de Teoría de la Señal y Comunicaciones e Ingeniería Telemática) y del grupo TRASGO (Departamento

de Informática) la Universidad de Valladolid ha recibido recientemente el reconocimiento "*GPU Research Center*" que otorga NVIDIA, y que certifica el empleo de CUDA y GPUs de Nvidia en programas de investigación. Este reconocimiento se suma al sello ya obtenido en 2014 de "*GPU Education Center*" que apoya y fomenta la enseñanza de la programación CUDA C/C++ como parte de su oferta educativa.

1.2. Fases y métodos

Las fases en las que se ha dividido el presente proyecto fin de carrera se pueden agrupar en tres categorías principales: estudio, desarrollo, y conclusiones y resultados. Estas fases de manera más detallada son las siguientes:

- Estudio de las arquitecturas GPU de NVIDIA GeForce Serie 7.
- Estudio del lenguaje de programación sobre GPUs de NVIDIA CUDA C. Para esto se ha empleado el libro "CUDA By Example (2011), Jason Saders y Edward Kadrot", así como los manuales oficiales gratuitos de NVIDIA: "NVIDIA CUDA C Programming Guide" y "CUDA C Best Practices Guide".
- Realización de ejemplos de aprendizaje de programación en CUDA C (ver apartado 2.5).
- Estudio del esqueleto de programa del que se partió. Comprensión y familiarización con el código.
- Modificación del código y adaptación para nuestro algoritmo de cálculo del producto escalar.
- Estudio de los ficheros c-mex, y creación de uno de estos para la ejecución de nuestro código desde MATLAB.
- Escritura de un script de MATLAB que lanzara las distintas pruebas realizando los cálculos en CPU y en GPU, y que escribiera los resultados en un fichero de texto.
- Estudio y valoración de los resultados obtenidos.

1.3. Medios disponibles

Este trabajo fin de grado se ha realizado en el seno del Grupo de Telemática e Imagen (GTI) de la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universidad de Valladolid. Los medios necesarios para la realización de este trabajo no son muy exigentes, ya que el requisito principal es un ordenador con una GPU de arquitectura NVIDIA con posibilidad de programación en CUDA.

1.3.1. Hardware:

Se ha utilizado el servidor *"dilbert"* del laboratorio del GTI que dispone del siguiente equipamiento hardware:

- Dos procesadores modelo Intel Xenon X5650 con las siguientes características:
  - Velocidad de reloj: 2,66 GHz
  - Número de núcleos: 6
- Memoria RAM: 24 GB
- Tarjeta gráfica modelo NVIDIA GeForce GTX 770 con las siguientes características:
  - Número de multiprocesadores (SM): 8
  - CUDA Cores: 1536
  - Frecuencia de reloj normal: 1046 MHz
  - Frecuencia de reloj acelerada: 1085Mhz
  - Velocidad de la memoria: 7,0 Gbps
  - Configuración de memoria estándar: 2048 MB GDDR5
  - Andcho de banda de memoria 225,3 GB/s (NVIDIA Corp., 2015c)
- Tarjeta gráfica modelo NVIDIA GeForce GTX 780 con las siguientes características:
  - Número de multiprocesadores (SM): 12
  - CUDA Cores: 2304
  - Frecuencia de reloj normal: 863 MHz
  - Frecuencia de reloj acelerada: 900 Mhz
  - Velocidad de la memoria: 6,0 Gbps
  - Configuración de memoria estándar: 3072 MB GDDR5
  - Andcho de banda de memoria 288,4 GB/s (NVIDIA Corp., 2015d)

1.3.2. Software:

A nivel de software las principales aplicaciones utilizadas para el desarrollo del trabajo han sido:

- Sistema operativo CenOS 6
- Nsight 5.5
- NVIDIA CUDA Toolkit 5.5
- Controlador NDVIDIA 331.38
- MATLAB R2013b
- Paquete servidor vnc-server

- TigerVNCViewer v1.4.0
- Putty 0.63

El NVIDIA CUDA Toolkit proporciona un entorno completo de desarrollo para programar en C y C++ utilizando la GPU para acelerar las aplicaciones. Este conjunto de herramientas incluye un compilador para GPUs NVIDIA, librerías matemáticas (no se han utilizado en este trabajo), y otras herramientas para el depurado y optimización del desempeño de las aplicaciones.

Las tres últimas aplicaciones software mencionadas se han utilizado debido a que no se ha trabajado directamente sobre el servidor "*dilbert*", sino que se ha hecho en remoto utilizando el software VNC (*Virtual Network Computing*). VNC permite mostrar una sesión "X Window" corriendo en otra computadora, esto es, que podemos manejar el servidor desde un equipo remoto a través de internet. Para ello hace falta un servidor VNC instalado en "*dilbert*", y un programa de visualización en la estación cliente en la que se muestra el display que está corriendo en el servidor. Debido al cortafuegos de la escuela que bloqueaba los servicios VNC fue necesario crear un túnel SSH mediante el programa Putty para encapsular la comunicación VNC entre cliente y servidor.

1.4. Organización del Trabajo Fin de Grado

El presente trabajo fin de grado se estructura en 4 capítulos, claramente diferenciados entre ellos por el contenido que tratan. A continuación describimos brevemente el contenido de cada uno de ellos:

- **Capítulo 2: Programación de propósito general en GPU a través de NVIDIA CUDA C.** Aquí se dará una breve visión de las técnicas de *GPU Computing* y programación paralela, además del lenguaje de programación NVIDIA CUDA C y otros aspectos relacionados con la programación en GPU. Como apéndice a este capítulo se añade una sección con unos ejemplos de aprendizaje.

- **Capítulo 3: Descripción del código.** Se explicarán las principales características del programa escrito, también se harán unos breves comentarios acerca de las funciones en las que se organiza el código.

- **Capítulo 4: Pruebas y resultados. Conclusiones y líneas futuras.** Se abordarán las diferentes pruebas realizadas sobre la implementación y se expondrán los resultados obtenidos. Además, se analizará las mejoras que proporcionan el empleo de la programación en GPU frente a CPU, el uso de streams y la utilización de multi-GPU. Por otra parte, se analizarán las conclusiones obtenidas después del

desarrollo explicado en los capítulos anteriores y se dará una visión de lo que podría hacerse en el futuro a partir de este trabajo.

Capítulo 2

Programación de propósito general en GPU a través de NVIDIA CUDA C

El objetivo principal de este capítulo es dar una visión general de la programación de propósito general sobre GPU a través del lenguaje NVIDIA CUDA C. Se explicará algunos de los conceptos más importantes de la programación paralela en general y sobre GPU en particular, debido a que no todos los programadores están relacionados con algunas de estas ideas. Se hará especial hincapié, entrando más en detalle, en aquellos aspectos que se utilizarán en el desarrollo del código (Capitulo 3). Al final de este capítulo se añade como apéndice una serie de ejemplos de aprendizaje de la programación en CUDA C.

2.1. Introducción

2.1.1. Arquitecturas Hardware: CPU/GPU

En el esquema genérico de la arquitectura de un ordenador (Figura 2.1) la tarjeta gráfica no está integrada en la placa base, se dice que tiene una tarjeta gráfica dedicada. La CPU se comunica a través del puente norte (*North Bridge*) de la placa base con la memoria RAM (*Random Access Memory*) y también con la GPU, conectada esta última mediante puertos AGP (*Accelerated Graphics Port*) o PCI Express (*Peripheral Component Interconnect Express*). El puente sur (*South Bridge*), también conocido como "concentrador de controladores de entrada/salida", se encarga de coordinar los diferentes dispositivos de entrada y salida, tales como USB, PCI, FireWire, etc.., y algunas otras funcionalidades de baja velocidad dentro de la placa base.

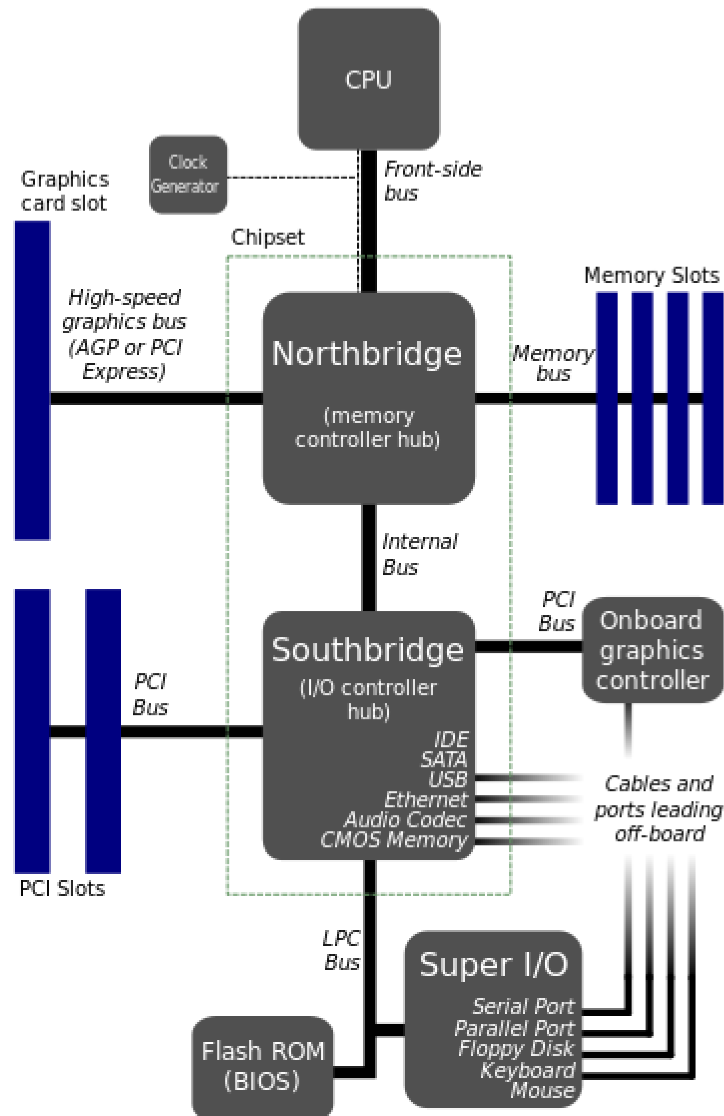


Figura 2.1: Esquema genérico del hardware de un PC (Fuente: https://commons.wikimedia.org/wiki/File:Motherboard_diagram.svg)

La misión principal de la GPU es liberar a la CPU de los costosos cálculos en coma flotante y renderizar descripciones 3D. Debido a esto y a la gran demanda de cómputo visual de los últimos años las GPUs se han convertido en potentes multiprocesadores con millones de transistores, superando tanto en número de procesadores como de transistores a las CPUs.

La GPU como procesador gráfico integrado en una tarjeta gráfica, puede comunicarse con la memoria local alojada en la propia tarjeta sin tener que hacer un uso compartido de la memoria RAM del sistema. Es importante notar las enormes diferencias en ancho de banda existente entre la GPU y la memoria gráfica local respecto al ancho de banda disponible en otras partes del ordenador.

Por otro lado las aplicaciones GPGPU pueden explotar ciertas características de la memoria de vídeo para aumentar más su rendimiento. El subsistema de memoria para texturas, por ejemplo, está optimizado para el acceso a patrones locales durante la renderización de triángulos, lo que hace a la GPU particularmente útil para aplicaciones en las que haya que realizar cálculos sobre mallas o matrices 2D (imágenes).

2.1.2. Sistemas de procesamiento secuencial y paralelo

La taxonomía de Flynn es una clasificación clásica de arquitecturas de ordenadores, publicada por primera vez en 1966, en la que se clasifican las distintas arquitecturas atendiendo a los flujos de instrucciones y de datos.

Se define como flujo de instrucciones al conjunto de instrucciones secuenciales que son ejecutadas por un único procesador, y como flujo de datos al flujo secuencial de datos requeridos por el flujo de instrucciones. Con estas consideraciones, Flynn clasifica los sistemas en cuatro categorías, como podemos observar en la Tabla 2.1.

	Flujo único de instrucciones	Flujos múltiples de instrucciones
Flujo único de datos	SISD	MISD
Flujos múltiples de datos	SIMD	MIMD

Tabla 2.1: Diferentes arquitecturas según la taxonomía de Flynn

Los sistemas SISD (*Single Instuction Single Data*) se caracterizan por tener un único flujo de instrucciones sobre un único flujo de datos, es decir, se ejecuta una instrucción detrás de otra. Este es el concepto de arquitectura serie descrita por Von Neumann donde, en cualquier momento, sólo se ejecuta una única instrucción sobre un único dato. Un ejemplo de estos sistemas son las máquinas secuenciales convencionales. El esquema de esta arquitectura es el que aparece en la Figura 2.2 (Tejero de Pablos, 2009).

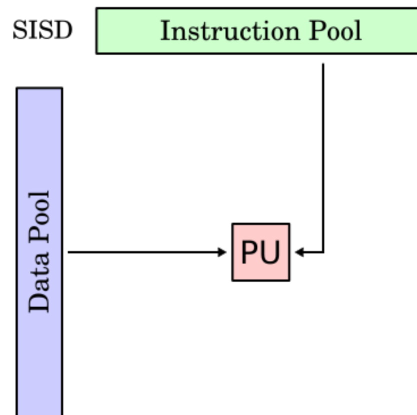


Figura 2.2: Sistema SISD (Single Instruction Single Data)

Los sistemas SIMD (*Single Instuction Multiple Data*) tienen un único flujo de instrucciones que operan sobre múltiples flujos de datos. Las unidades de ejecución responden a una única instrucción que aplican sobre una gran cantidad de datos (concurrencia de operación), de ahí que sea conveniente que exista paralelismo entre los datos, y se considere a este tipo de arquitectura como el origen de la máquina paralela. Las distintas instrucciones se siguen sucediendo de manera secuencial (Hennessy y Patterson, 1993).

El funcionamiento de este tipo de sistemas es el siguiente: la Unidad de Control manda una misma instrucción a todas las unidades de proceso (ALUs). Las unidades de proceso operan sobre datos diferentes pero con la misma instrucción recibida. El esquema se muestra en la Figura 2.3, y algunos ejemplos de estos sistemas los tenemos en las máquinas vectoriales, y en las GPUs originales (Tejero de Pablos, 2009).

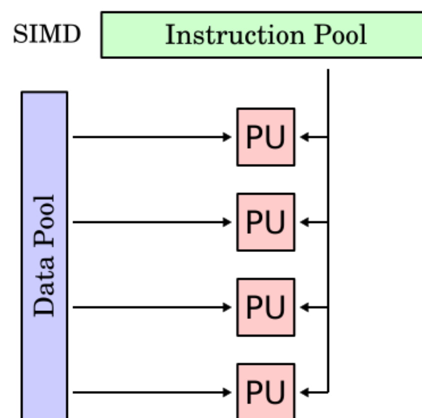


Figura 2.3: Sistema SIMD (Single Instruction Multiple Data)

Los sistemas MISD (*Multiple Instuction Single Data*) poseen múltiples instrucciones que operan sobre un único flujo de datos. Estos sistemas se pueden contemplar de dos maneras distintas: varias instrucciones operando simultáneamente sobre un único dato o varias instrucciones operando sobre un dato que se va convirtiendo en un resultado que será la entrada para la siguiente etapa. En el primer caso se trabaja de forma segmentada, luego todas las unidades de proceso pueden trabajar de forma concurrente.

El esquema general de estos sistemas se muestra en la Figura 2.4., y aunque no tienen muchas aplicaciones prácticas, algunos ejemplos de estos tipos de sistemas son los arrays sistólicos o arrays de procesadores (Tejero de Pablos, 2009).

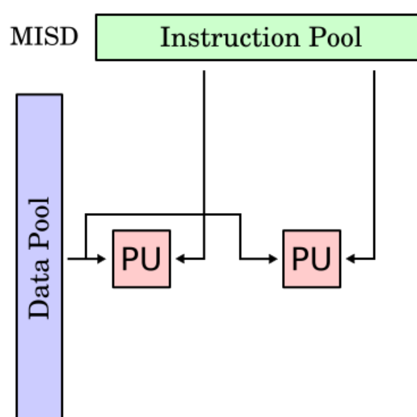


Figura 2.4: Sistema MISD (Multiple Instruction Single Data)

Los sistemas MIMD (*Multiple Instuction Multiple Data*) tienen varios procesadores que funcionan de manera asíncrona e independiente, en cualquier momento, cualquier procesador puede ejecutar diferentes instrucciones en diferentes datos. Se utilizan estas arquitecturas cuando se persigue algún tipo de paralelismo.

Según su acceso a memoria, se distinguen dos tipos de máquinas MIMD: los sistemas de memoria compartida, que ofrecen al programador una sola dirección de memoria a la que todos los procesadores pueden acceder; y los sistemas de memoria distribuida, en los que será necesario la comunicación entre procesadores para compartir datos (Hennessy y Patterson, 1993). El esquema de estas arquitecturas se muestra en la Figura 2.5, y el ejemplo más claro de este tipo de sistemas son los procesadores con varios núcleos (*multi-core*).

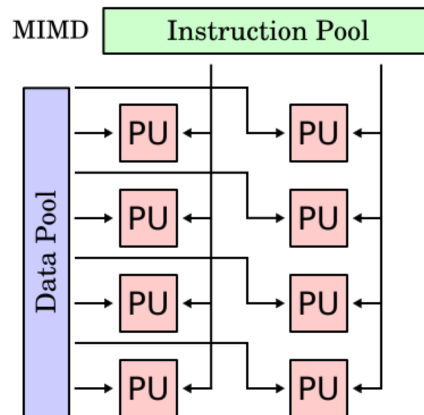


Figura 2.5: Sistemas MIMD (Multiple Instruction Multiple Data)

2.1.3. Modelos de computación

El modelo de Von Neumann, véase Figura 2.6, es utilizado clásicamente en el diseño de las CPUs, en este modelo las instrucciones se almacenan como datos y sus operadores definen la forma en que estos se localizan dentro del hardware (Casado García, 2010).

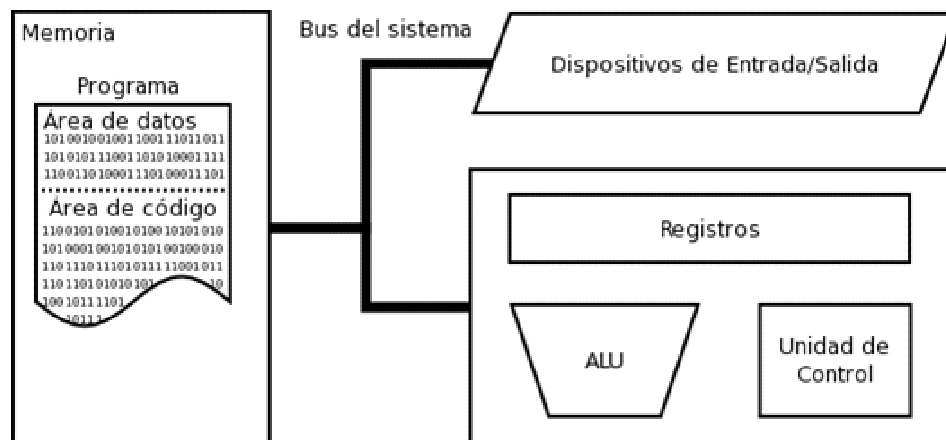


Figura 2.6: Modelo Von Neumann

El principal cuello de botella del PC es el acceso a memoria, ya que por cada instrucción a ejecutar (por ejemplo una suma) se requieren 4 datos: 2 operandos origen, la operación (suma) y el destino. Por lo tanto los grandes fabricantes han dedicado muchos esfuerzos en el desarrollo del acceso a memoria, mejorando caches, comunicaciones, etc.

La clave para entender el cuello de botella del modelo convencional es que el desarrollo no se debe centrar en reducir la latencia del acceso individual a memoria, sino en maximizar el ancho de banda, que será lo que dé fuerza al procesador cuando tenga una gran cantidad de datos que procesar (Tejero de Pablos, 2009).

Una alternativa al modelo de Von Neumann es el modelo *Streaming*, véase Figura 2.7, que posee una arquitectura que pretende subsanar el cuello de botella a la hora de acceder a los datos que presenta el anterior modelo. En vez de desplazar las instrucciones por el cauce segmentado del procesador y desplazar los datos a su encuentro, resulta más lógico dejar que circulen los datos, que constituyen la mayor parte de la aplicación, y que se dirijan hacia los operadores que son pocos y simples.

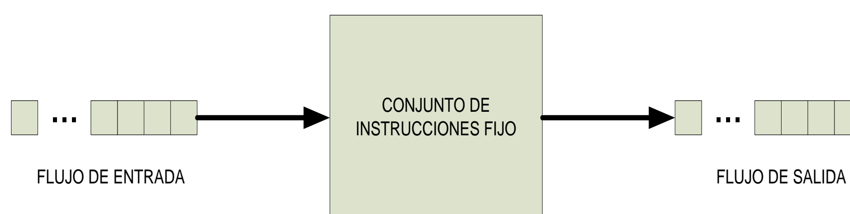


Figura 2.7: Modelo Streaming

Con este modelo se consigue dar la vuelta al embudo colocando la parte más ancha del lado de los datos y la más estrecha del lado de las instrucciones. Ahora, los datos se encuentran circulando de forma permanente dentro de la arquitectura, y las instrucciones definen la forma en que éstos son transformados a medida que avanzan.

Como resultado, se tiene un *streamprocessor* o procesador circulante, un sistema mucho más compensado para aplicaciones gráficas, pues explota la potencia del hardware proporcionalmente a como se necesita. En principio, el precio a pagar en el modelo *Streaming* es la generalidad que envuelve a la arquitectura: una CPU ha venido utilizándose para ejecutar cualquier tipo de algoritmo, pero una GPU sólo podrá ser utilizada en algoritmos que pueden expresarse mediante el modelo *Streaming*. Esto es algo restrictivo, por lo que puede requerirse una modificación importante del algoritmo para su adaptación a este modelo (Casado García, 2010).

2.1.4. Evolución de la graphics pipeline en GPU

En la Figura 2.8 se muestra el esquema de la arquitectura de las serie Geforce 7, basada en la *graphics pipeline*, que es el camino que siguen los datos en la GPU. Estos datos pasan por módulos, algunos de ellos programables como el *vertex processor* y el *fragment*

processor, donde cada uno ejecuta distintas operaciones sobre los datos. Se tratan de módulos dedicados, como el *vertex processor* que ejecuta operaciones sobre vértices y el *fragment*, que ejecuta operaciones sobre fragmentos (Casado García, 2010).

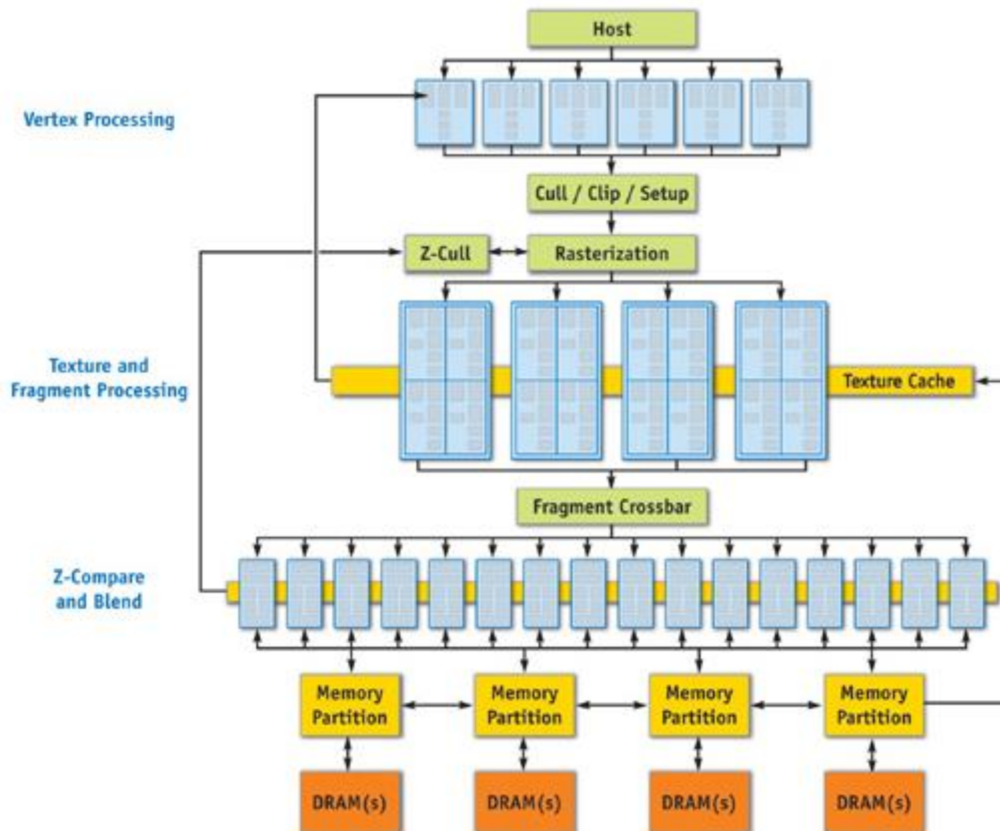


Figura 2.8: Arquitectura de la serie GeForce 7 de NVIDIA (Fuente: Pharr, 2005)

En la Figura 2.9 se puede observar un esquema de la arquitectura de la serie NVIDIA GeForce 8. En este esquema cabe destacar la realimentación de los procesadores en la pipeline, esto es porque ya no existe el concepto de *vertex* y *fragment processors* sino que son los mismos procesadores los que realizan ambas funciones (arquitectura de *shaders* unificados). Por tanto, las distintas etapas ejecutadas por varios módulos se reúnen en una única.

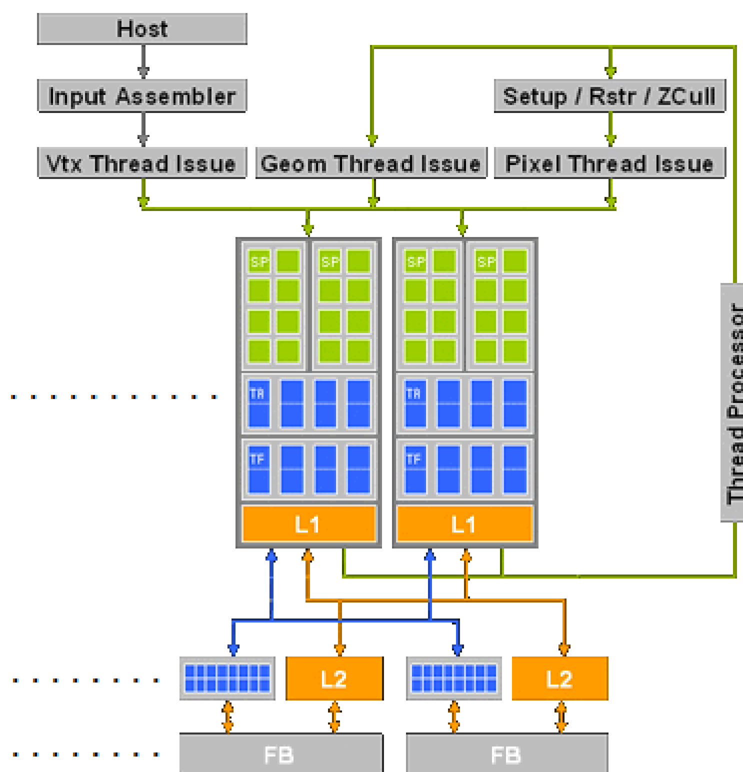


Figura 2.9: Arquitectura de la serie GeForce 8 de NVIDIA (Fuente Pharr, 2005*)*

Son muchas las ventajas de esta nueva arquitectura. Por ejemplo, el antiguo problema de cambiar constantemente la carga de trabajo y una etapa de *shader* creando un cuello de botella de procesamiento es resuelto ya que las unidades pueden adaptarse dinámicamente, ahora que están unificadas.

Es el comienzo del concepto de procesadores genéricos en la GPU, que precede a la aparición de NVIDIA CUDA.

2.2. Computación de propósito general en GPU

Las vías de investigación para la mejora de las arquitecturas de computación se encuentran en un punto de cambio. En los últimos años las velocidades de reloj de las CPUs prácticamente han parado de incrementarse, debido principalmente a los problemas de consumo, disipación de calor y barrera energética. Es por ello que ahora los principales fabricantes de CPUs se están centrando en el desarrollo de procesadores con varios núcleos (*Cores*) en lugar de intentar aumentar la velocidad de reloj de las CPUs.

En la actualidad, las principales mejoras en el rendimiento de los procesadores vienen del paralelismo y no del incremento de la velocidad con un único procesador. En cambio los procesadores gráficos (GPUs) ya contienen múltiples unidades de procesamiento paralelo que son capaces de, en ciertas aplicaciones, dar un rendimiento mucho mayor que las CPUs más modernas (NVIDIA Corp., 2015).

GPGPU es un término que hace referencia al uso de las GPUs como procesadores de propósito general, actualmente llamado *GPU Computing*. Aunque estos procesadores están diseñados para actuar como procesadores de apoyo en aplicaciones gráficas, liberando a la CPU del procesado de gráficos, son muy útiles para otras aplicaciones, ya que se comportan como coprocesadores paralelos, a nivel de datos, para la CPU. Algunas de las principales razones por las que en la actualidad se está difundiendo el uso de estos procesadores para cálculos diferentes al procesado de gráficos son las siguientes:

- Son procesadores masivos de streams que pueden conseguir superar ampliamente el rendimiento de las CPUs en aplicaciones que requieran paralelismo.
- Soportan variables en punto flotante de 32 bits y 64 bits.
- Tienen un modelo de programación relativamente flexible.
- Disponen de un enorme ancho de banda para el acceso a memoria.

2.2.1. APIs gráficos y lenguajes de programación

La programación de aplicaciones gráficas ha cambiado mucho en los últimos años. En la década de los ochenta, se utilizaba un programa en C con llamadas a servicios de interrupción para operaciones gráficas.

A mediados de los 90, tomaron el relevo DirectX y OpenGL; con una dualidad muy similar a la de Windows y Linux en los sistemas operativos. Estas API pueden verse como un conjunto de librerías que se invocan desde un programa C igual que cualquier rutina o procedimiento, proporcionando una serie de operadores gráficos que se ejecutan sobre la mayoría de las tarjetas gráficas. De esta manera, actúan como una capa intermedia que hace al software transparente a los cambios del hardware.

En esta evolución del hardware se encuentra uno de los principales avances en tarjetas gráficas, la inclusión de procesadores programables dentro de la GPU: uno dedicado al procesamiento de vértices y otro al de píxeles.

A partir de aquí, la versión 9.0 de DirectX extendió en 2002 la funcionalidad de estos dos procesadores, incorporando instrucciones de salto condicional en el procesador de vértices y coordenadas de texturas y computación de punto flotante en el procesador de píxeles. Los aditivos hardware fueron proporcionados por NVIDIA y, respecto al software,

Microsoft desarrollo el lenguaje HLSL (*High Level Shader Language*), lenguaje de sombreado para su uso con la API de Microsoft, Direct3D.

Dado que ambas firmas estaban interesadas en la programación de estos procesadores, Microsoft y NVIDIA trabajaron conjuntamente para desarrollar el lenguaje de alto nivel Cg (*C for Graphics*), con las mismas funcionalidades de HLSL pero adoptando la sintaxis al lenguaje C. En paralelo con el desarrollo de Cg se comenzó a desarrollar GLSL (*OpenGL Shading Language*), tecnología que parte del API estándar OpenGL para permitir especificar segmentos de programas gráficos que serán ejecutados sobre la GPU (Casado García, 2010).

Dentro de las APIs clásicas, en los que el software son las aplicaciones gráficas y el hardware es la arquitectura PC disponemos de dos grandes planteles: OpenGL y Direct3D.

- OpenGL (*Open Graphics Library*): especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.
- Direct3D: es parte de DirectX, y se trata de una API propiedad de Microsoft para la programación de gráficos 3D.

Sin embargo, en la actualidad el futuro parece llevar a arquitecturas más heterogéneas, donde la programación se lleve a cabo de una forma más genérica, al igual que ocurre en CPU. Un ejemplo de esto es CUDA C, variación del lenguaje de programación C para codificar algoritmos en GPUs de NVIDIA, y otro es OpenCL (Open Computing Language) (Casado García, 2010).

2.3. NVIDIA CUDA

2.3.1. Introducción

2.3.1.1. Uso de la GPU para computación de datos en paralelo

Debido a una insaciable demanda de mercado en gráficos 3D de alta definición, la tecnología de las GPUs ha evolucionado hacia un alto grado de paralelismo, con un elevado número de procesadores con una gran potencia de cálculo y un ancho de banda de memoria muy alto. Esta tendencia se aprecia claramente en las figuras 2.9 y 2.10 (NVIDIA Corp., 2015a).

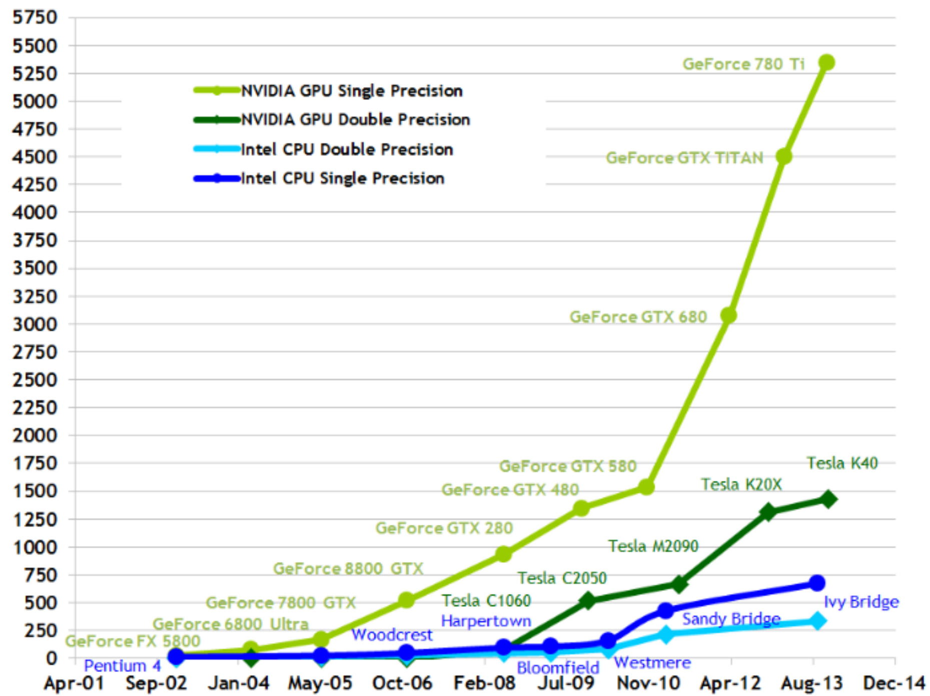


Figura 2.9: Número de operaciones en punto flotante por segundo para la CPU y la GPU

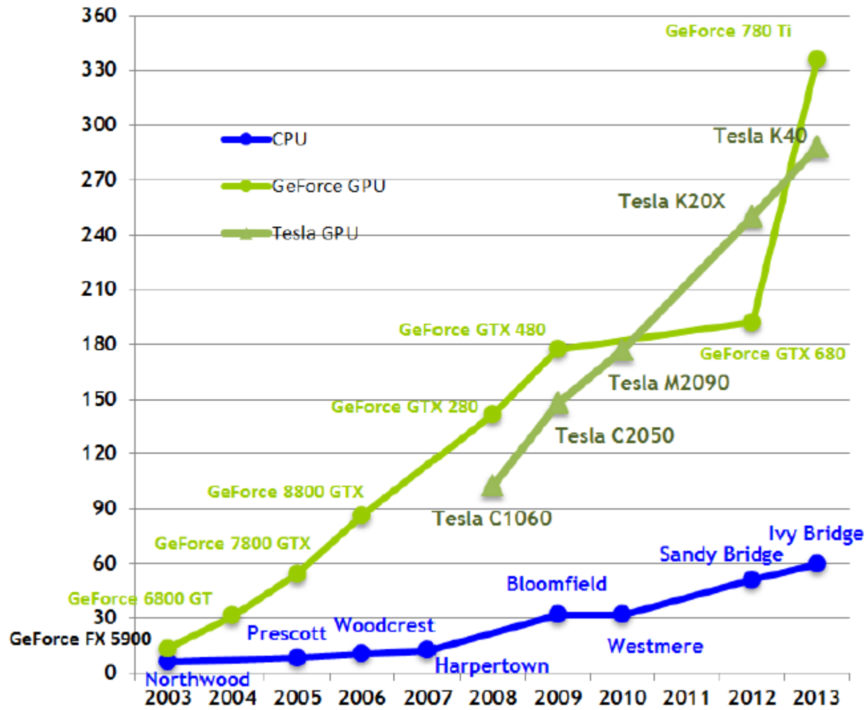


Figura 2.10: Ancho de banda en memoria para la CPU y la GPU

La razón de esta diferencia de capacidad de cálculo en operaciones de punto flotante entre la CPU y la GPU, reside en que esta segunda está especializada en tareas de cálculo intenso y por tanto diseñada de tal manera que la mayor parte de transistores están dedicados al procesamiento de datos (ALU), en lugar de almacenamiento en memoria cache o control de flujo. Figura 2.11 (NVIDIA Corp., 2015a).

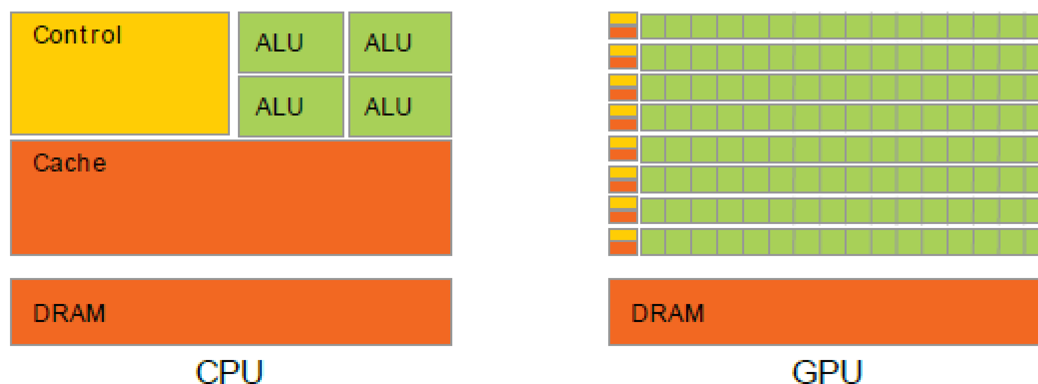


Figura 2.11: Espacio del chip dedicado a memoria y cálculo en la CPU y la GPU

Más específicamente, la GPU se ajusta especialmente bien a problemas que pueden ser expresados algorítmicamente como cálculos paralelos (los datos no dependen unos de otros), en los que el mismo programa es ejecutado sobre muchos elementos en paralelo, y con una intensidad aritmética mucho mayor que las operaciones de memoria. De manera esquemática esto nos lleva a que ejecutando un solo programa en GPU se nos lancen automáticamente una serie de hilos que manejaran datos distintos del programa, realizarán los cálculos oportunos sobre ellos y nos devolverán los resultados oportunos, consiguiendo realizar más cálculos en un menor tiempo. Como el mismo programa es ejecutado para cada elemento, hay un bajo requerimiento de control de flujo sofisticado; y cuando la intensidad aritmética es elevada, la latencia de los accesos a memoria puede ser escondida con cálculos en lugar de grandes cachés de datos (García Casado, 2010).

El procesamiento de datos en paralelo mapea elementos de datos en hilos de procesamiento en paralelo. Muchas aplicaciones que procesan grandes conjuntos de datos como arrays pueden usar un modelo de programación en paralelo para acelerar las computaciones. Por ejemplo las renderizaciones 3D de grandes conjuntos de pixeles y vértices son mapeadas en hilos paralelos. Igualmente, aplicaciones de procesamiento de imagen como el post-procesado de imágenes renderizadas, codificación y decodificación de vídeo, escalado de imagen y reconocimiento de patrones puede mapear bloques y píxeles de imagen en hilos de procesamiento en paralelo. De hecho, muchos algoritmos fuera del campo de la imagen son acelerados por procesamiento de datos en paralelo, desde el

procesamiento general de señales o simulaciones físicas hasta finanzas computacionales o biología computacional (Por ejemplo: Xiong, 2012 y Trapnell y Schatz 2009).

Hasta ahora, acceder a todo este poder computacional acumulado en la GPU y dedicarlo para aplicaciones no gráficas seguía siendo problemático:

- La GPU sólo podía ser programada a través de APIs gráficas que conllevaban una alta curva de aprendizaje.
- La DRAM de la GPU podía ser leída de forma general pero no podía ser escrita de forma general.
- Algunas aplicaciones sufrían cuellos de botella por el ancho de banda de la DRAM.

En noviembre de 2006 NVIDIA lanza CUDA C, un modelo de programación para un nuevo hardware que soluciona estos problemas y permite utilizar la GPU como un auténtico dispositivo de computación genérica de datos en paralelo.

2.3.1.2. Arquitectura NVIDIA CUDA

CUDA es una arquitectura utilizada para realizar computaciones en la GPU como un dispositivo de computación de datos en paralelo sin la necesidad de mapearlos en una API gráfica. Se encuentra disponible desde la serie GeForce 8 de NVIDIA (la primera tarjeta en implementar el modelo CUDA fue la GeForce 8800 GTX). El mecanismo multitarea del sistema operativo, el cual permite que varios procesos sean ejecutados al mismo tiempo, es responsable de controlar el acceso a la GPU por varias aplicaciones de CUDA y gráficas corriendo concurrentemente.

La pila software de NVIDIA CUDA está compuesta, como se muestra en la Figura 2.12, por muchas capas: un controlador hardware, una API y su *runtime*, y dos librerías matemáticas de más alto nivel de uso común. El hardware ha sido diseñado para soportar controladores ligeros y capas de *runtime*, y dar como resultado un alto rendimiento (NVIDIA Corp., 2009a).

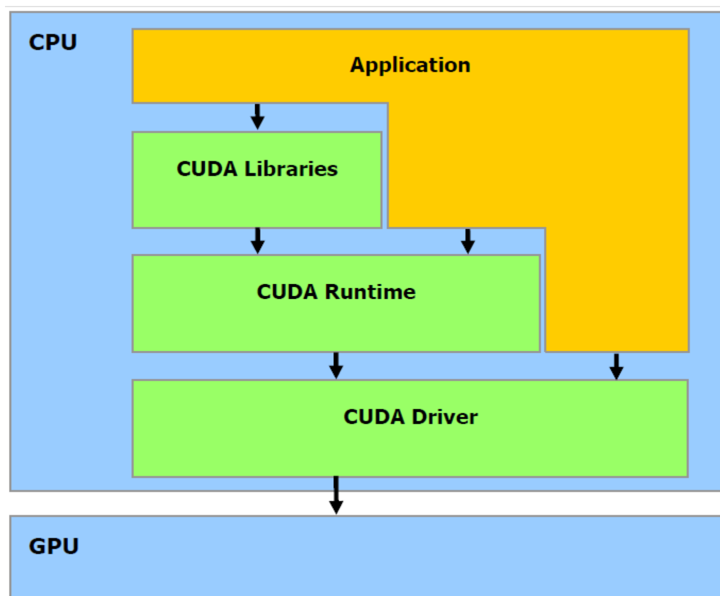


Figura 2.12: Pila software de CUDA

A diferencia de generaciones previas de GPUs que dividían los recursos de computo entre *vertex* y *pixel shaders* la arquitectura CUDA incluye una tubería *shader* (*shader pipeline*) unificada, permitiendo que todas y cada una de las ALUs del chip se puedan utilizar para cálculos de propósito general. Esto a su vez, condiciona a que las ALUs se construyan respetando los requisitos IEEE para operaciones de punto flotante y que utilicen un conjunto de instrucciones para cálculo general en lugar de uno específico para gráficos. Además se permite a las unidades de ejecución de la GPU lecturas y escrituras de acceso arbitrario en memoria (respectivas operaciones de *gather* y *scatter* de datos en posiciones dispersas), y se provee de un controlador software para el acceso a memoria cache conocido como memoria compartida (*shared memory*) (Sanders y Kandrot, 2010).

2.3.2 Modelo de programación

2.3.2.1. Modelo de programación escalable

La llegada de las CPU *multicore* (varios núcleos) y las GPUs (*manycore*) supone que hoy en día la mayor parte de los procesadores son sistemas paralelos. Con el paso del tiempo el número de procesadores irá incrementando, como ya se ha visto en años precedentes y como afirma la ley de Moore, luego el reto es desarrollar aplicaciones software que escalen transparentemente su paralelismo, para así obtener ventaja del incremento del número de procesadores (NVIDIA Corp., 2015a).

El modelo de programación CUDA proporciona tres conceptos o abstracciones claves con este fin: la jerarquía de grupos de hilos, las memorias compartidas y los puntos de sincronización. Estas abstracciones proporcionan distintos niveles de paralelismo, que permiten al programador dividir el problema en sub-problemas que se hacen automáticamente escalables. Es decir, estos sub-problemas se adaptan al número de multiprocesadores disponibles en cada GPU, y se deja como tarea del sistema en ejecución (*runtimesystem*) conocer el número de multiprocesadores. Figura 2.13 (NVIDIA Corp., 2015a).

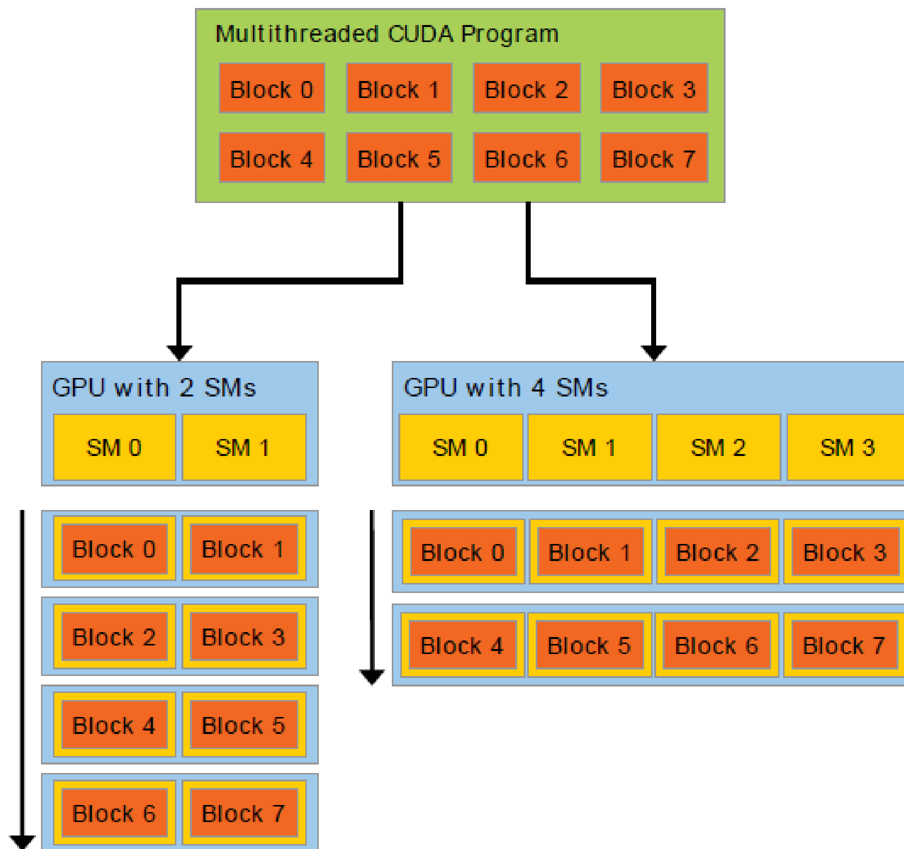


Figura 2.13: Escalado automático

Una GPU se construye sobre un array de *Streaming Multiprocessors* (SMs) (ver apartado implementación hardware). Un programa multihilo es dividido en bloques de hilos que se ejecutan independientemente unos de otros, de esta manera una GPU con más multiprocesadores ejecutará automáticamente un programa en menos tiempo que una GPU con menos.

A continuación pasamos a introducir los conceptos principales que residen detrás del modelo de programación CUDA.

2.3.2.2. La GPU como coprocesador multi-hilo de la CPU

En el modelo de programación, la GPU (*device*) es vista como un dispositivo de computación capaz de ejecutar un elevado número de hilos en paralelo, que opera como un coprocesador de la CPU principal (*host*). Luego las porciones de aplicaciones de computación intensiva de datos en paralelo corriendo en el host serán descargadas en el *device*, para que se ejecuten más rápidamente en el mismo.

Esto quiere decir que, cuando tenemos una porción de una aplicación que es ejecutada muchas veces pero independientemente sobre datos diferentes, puede ser aislada en una función que es ejecutada en paralelo en el dispositivo por N hilos diferentes. Para ello, dicha función es compilada con el conjunto de instrucciones del dispositivo y el programa resultante, llamado kernel, es descargado al dispositivo.

Ambos, el *host* y el *device* mantienen su propia DRAM, referida como memoria de host y memoria de *device* respectivamente. Una puede copiar datos desde una DRAM a la otra a través de llamadas API optimizadas que usan DMA (*Direct Memory Access*) (Casado Garcia 2010).

2.3.2.3. Agrupación de hilos en la GPU

Un lote de hilos que ejecuta un kernel se organiza como un *grid* (malla) de bloques de hilos como se describe en la Figura 2.14 (NVIDIA Corp., 2009). Un bloque de hilos es una agrupación de hilos que pueden cooperar juntos compartiendo datos eficientemente mediante memoria compartida rápida (*shared memory*) y sincronizando su ejecución para coordinar accesos a memoria y asegurar consistencia en los datos. De forma más precisa, se pueden especificar puntos de sincronización en el kernel, donde los hilos de un bloque son suspendidos hasta que todos alcanzan dicho punto. Para ello puede utilizarse la función `__syncthreads()`, invocable sólo en puntos del programa no divergentes (p. ej. no dentro de estructuras de control `if-else`) (NVIDIA Corp., 2009).

Cada hilo es identificado por su *thread ID*, que es el número de hilo dentro del bloque. Para ayudar con el direccionamiento complejo basado en este identificador (ID), una aplicación puede también especificar un bloque como un array de dos o tres dimensiones de tamaño arbitrario e identificar cada hilo usando índices de 2 o 3 componentes en su lugar. Para un bloque de dos dimensiones (Dx, Dy), el ID del hilo de índice (x, y) es $(x + y \cdot Dx)$ y para un bloque de tres dimensiones de tamaño (Dx, Dy, Dz) el ID de un hilo de índice (x, y, z) es $(x + y \cdot Dx + z \cdot Dx \cdot Dy)$.

Hay un número máximo limitado de hilos que un bloque puede contener, por ejemplo: en las GPUs con capacidad de cálculo 3.0 es 1024 hilos. Sin embargo, bloques con

las mismas dimensiones y tamaño que ejecutan el mismo kernel pueden ser agrupados en un *grid* de bloques, por lo que el número total de hilos que puede ser lanzado en una única invocación al kernel es mucho mayor.

Cada bloque es identificado por su block ID, que es el número de bloque dentro del *grid*. Para ayudar con el direccionamiento complejo basado en este ID, una aplicación puede también especificar un *grid* como un array de dos dimensiones de tamaño arbitrario e identificar cada bloque usando un índice de dos componentes en su lugar. Para un tamaño de bloque bidimensional (Dx, Dy), el ID del bloque de índice (x, y) es (x + y · Dx).

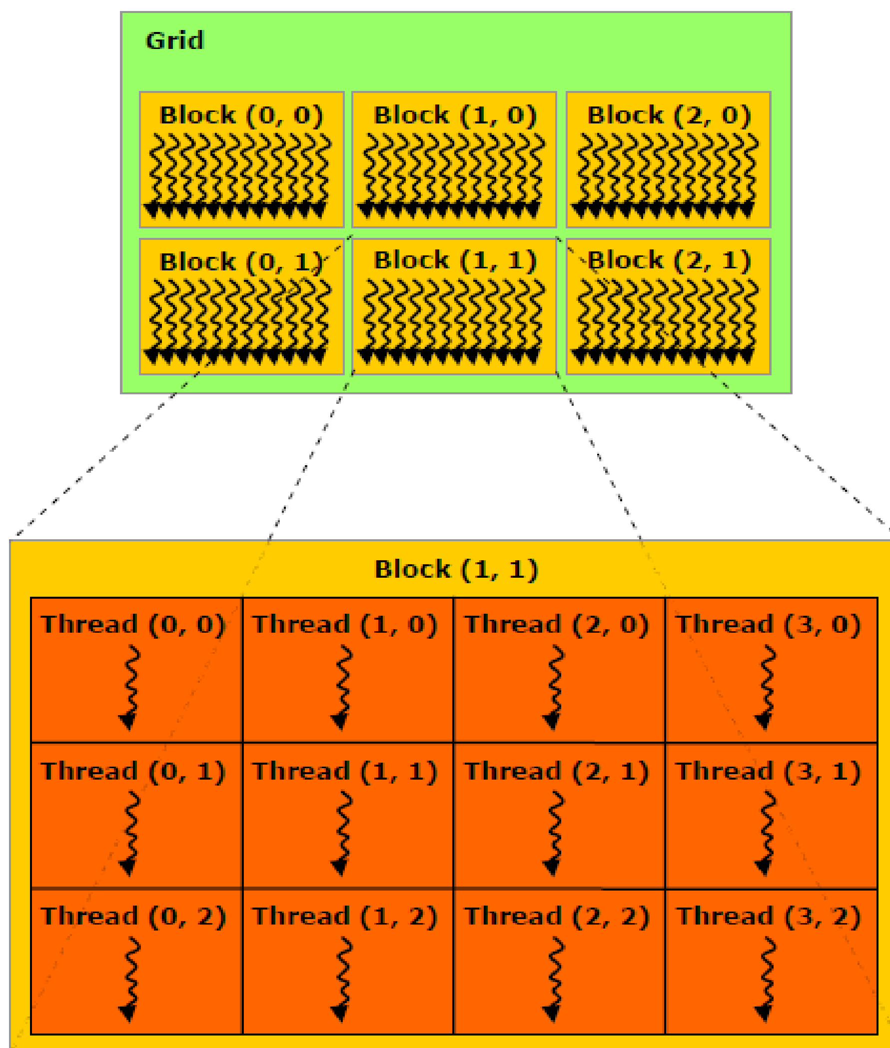


Figura 2.14: Jerarquía de hilos dentro de la GPU. Una malla de bloques de hilos

2.3.2.4. Modelo de memoria en GPU

Un hilo que se ejecuta en el *device* tiene sólo acceso a la memoria DRAM del *device* y a la memoria en chip a través de los siguientes espacios de memoria, como se muestra en la Figura 2.15 (NVIDIA Corp., 2009).

- Registros de lectura-escritura por hilo.
- Memoria local de lectura-escritura por hilo.
- Memoria compartida de lectura-escritura por bloque.
- Memoria global de lectura-escritura por *grid*.
- Memoria constante de sólo lectura sólo por *grid*.
- Memoria de textura de sólo lectura sólo por *grid*.

Los espacios de memoria global, constante y de textura pueden ser leídos o escritos por el host y son persistentes entre los lanzamientos de los kernel de una misma aplicación (NVIDIA Corp., 2009).

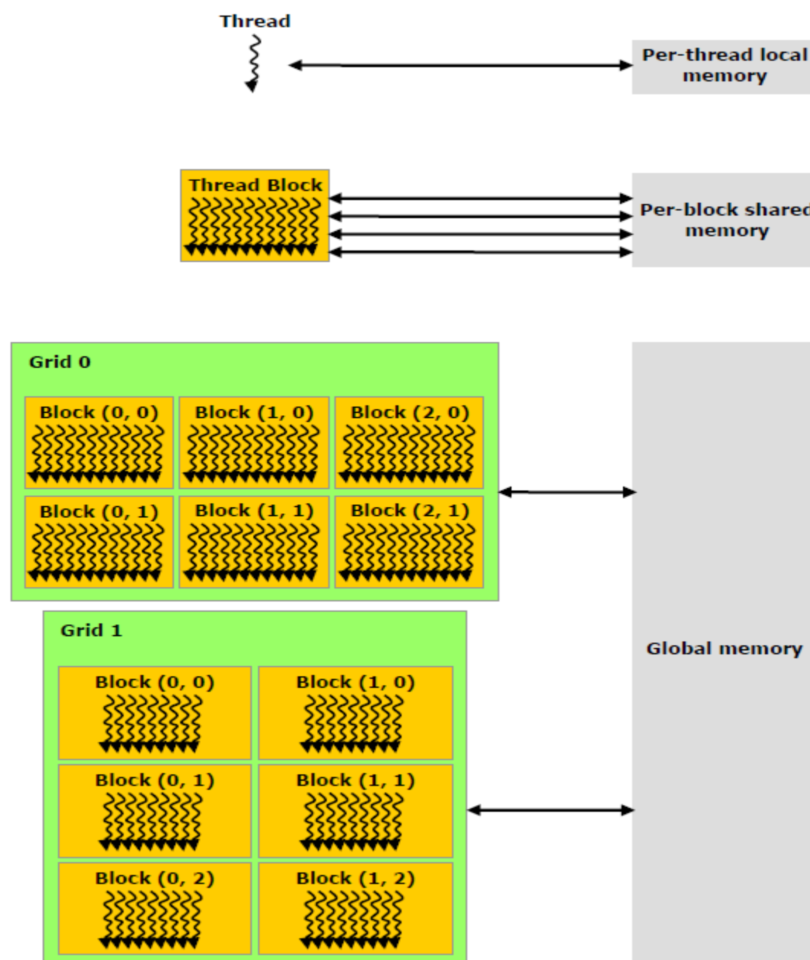
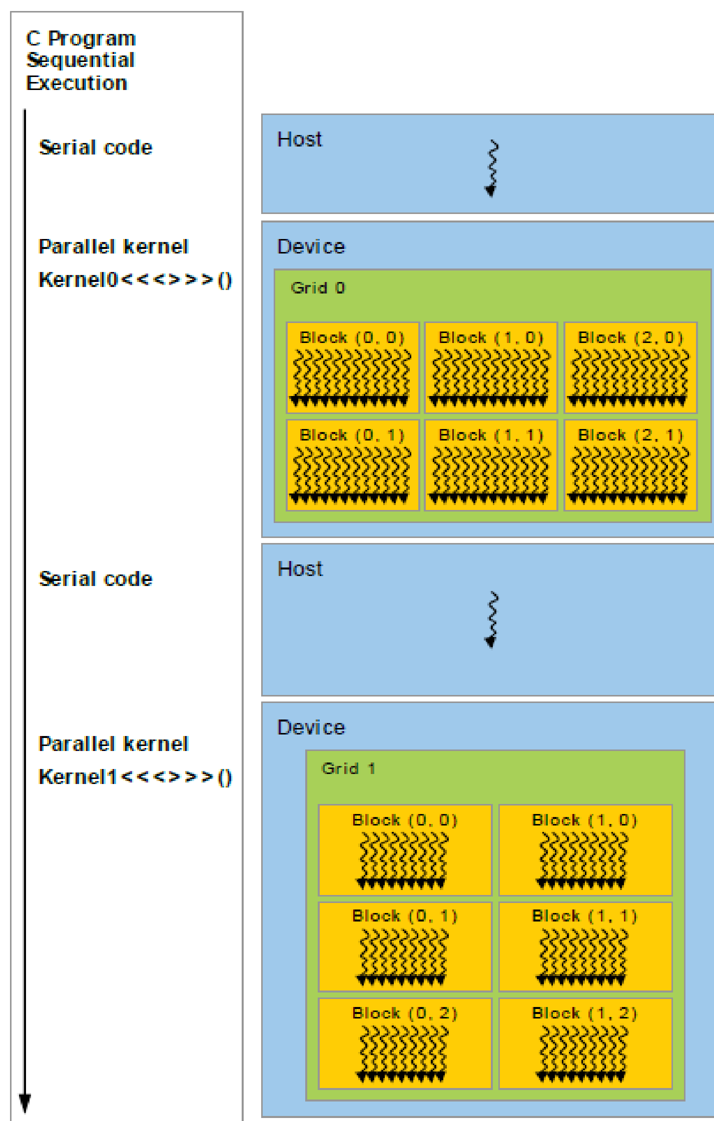


Figura 2.15: Jerarquía de memoria en la GPU

2.3.2.5. Host and Device

El modelo de programación de CUDA es un modelo de programación heterogénea, que asume que los hilos de CUDA son ejecutados en un dispositivo físicamente separado que opera como un coprocesador al host que ejecuta el programa en C. Este es el caso, por ejemplo, de los kernels que son ejecutados en una GPU y el resto del programa en C es ejecutado en una CPU, véase Figura 2.16 (NVIDIA Corp., 2015a).

El modelo de programación de CUDA C también asume que tanto el host como el *device* mantienen su propia DRAM, referida como *host memory* y *device memory*, respectivamente.



Serial code executes on the host while parallel code executes on the device.

Figura 2.16: Modelo de programación heterogénea, ejecución en CPU y en GPU

2.3.3. Implementación hardware de la GPU

La arquitectura de las GPUs de NVIDIA está construida alrededor de un array multihilo escalable de *Streaming Multiprocessors* (SMs) (Figura 2.17). Cada uno de estos multiprocesadores tiene en chip los siguientes cuatro tipos de memoria:

- Un set de registros locales de 32 bits por procesador.
- Una caché de datos paralela o memoria compartida que es compartida por todos los procesadores e implementa el espacio de memoria compartida.
- Una caché constante de sólo lectura que es compartida por todos los procesadores y acelera las lecturas desde el espacio de memoria compartida, que es implementada como una región de sólo lectura de la memoria del dispositivo.
- Una caché de textura de solo lectura que es compartida por todos los procesadores y acelera las lecturas desde el espacio de memoria de textura, que es implementado como una región de sólo lectura de la memoria del dispositivo.

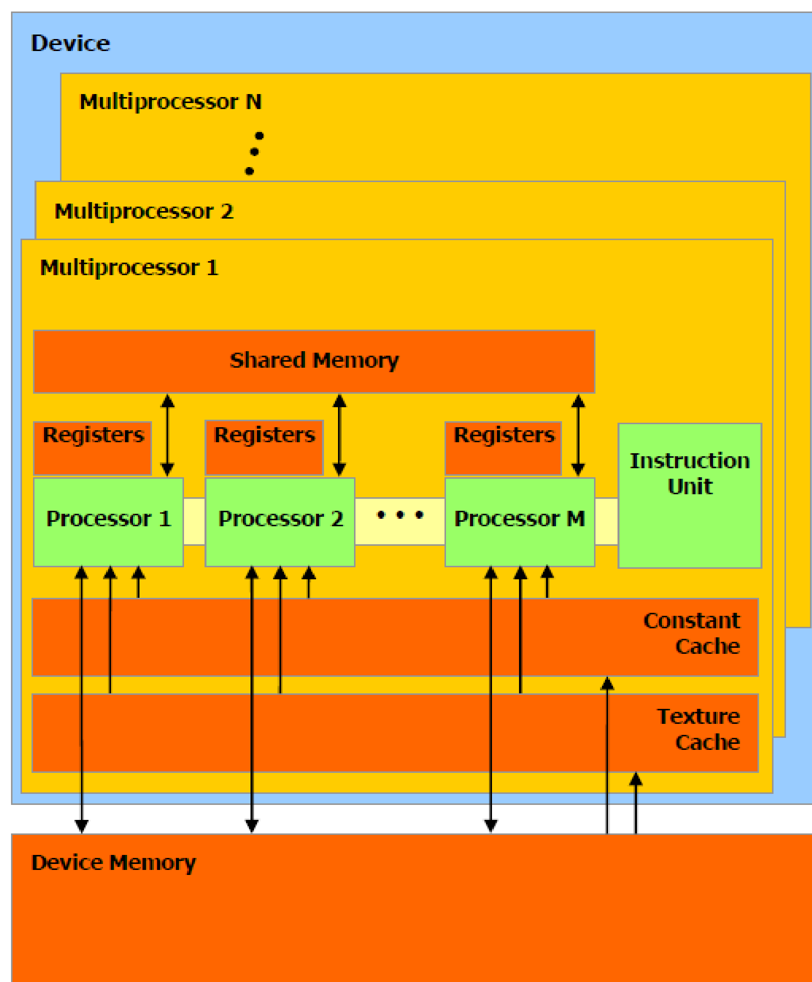


Figura 2.17: Modelo hardware de la GPU

Cuando un programa en CUDA C invoca desde la CPU (*host*) un kernel, los bloques de la malla de dicho kernel se enumeran y son distribuidos a los distintos multiprocesadores con capacidad de ejecución disponible. Los hilos de un mismo bloque se ejecutan simultáneamente en un multiprocesador, y varios bloques pueden ejecutarse también simultáneamente en el mismo multiprocesador. Cuando estos bloques han terminado otros nuevos bloques ocupan su lugar para ejecutarse (NVIDIA Corp., 2015a).

Los multiprocesadores para manejar el gran número de hilos que ejecutan simultáneamente (cientos de ellos) utilizan una arquitectura llamada SIMT (*Single Instruction, Multiple Thread*), que se podría considerar un híbrido entre las arquitecturas descritas más arriba: SIMD Y MIMD (simétrico).

2.3.3.1. Arquitectura SIMT: implementación hardware

Los multiprocesadores crean, gestionan, planifican y ejecutan los hilos en grupos de 32. Estos grupos son llamados *warps*. La ejecución de los 32 hilos se hace de manera simultánea y solo realizan una única instrucción en cada ciclo de reloj (esta es la parte del modelo SIMD).

La forma en la que un bloque está dividido en *warps* es siempre la misma; cada *warp* contiene hilos con threadIDs consecutivos e incrementales, con el primer *warp* conteniendo el hilo 0. El orden de los *warps* en un bloque no está definido, pero su ejecución puede ser sincronizada para coordinar accesos a memoria global o compartida.

El orden de los bloques en un *grid* de bloques de hilos no está definido y no hay un mecanismo de sincronización entre bloques, por lo que hilos de dos bloques diferentes del mismo *grid* no pueden comunicarse entre ellos a través de memoria global durante la ejecución del *grid*.

Cuando un SM recibe uno o más bloques de hilos para ejecutarlos, el SM lo divide en *warps* y el planificador de *warps* (*warp scheduler*) seleccionará uno de estos *warp* listos para ejecutarse.

Un *warp* ejecuta una única instrucción en cada momento, y entonces se alcanza la máxima eficiencia si los 32 hilos del *warp* coinciden en el mismo camino de ejecución, es decir, que no tienen diferentes vías de divergencia o ramificaciones (saltos condicionales). Si esto ocurre, aquellos hilos que no cumplen la condición serán marcados para permanecer inactivos y deberán esperar, y esto provoca ralentizaciones (NVIDIA Corp., 2015a).

En los casos en los que los hilos de un *warp* hacen una petición de acceso a memoria, para ocultar la latencia, inmediatamente después el planificador ejecutará en el siguiente

ciclo de reloj otro conjunto de hilos para aprovechar al máximo los recursos. La división de los bloques de hilos en *warp* se realiza simplemente de manera consecutiva.

2.3.3.3. Capacidad de cómputo

La capacidad de cómputo de un dispositivo es definida por un número de revisión mayor y un número de revisión menor.

- Los dispositivos con el mismo número de revisión mayor pertenecen a la misma arquitectura central.
- El número de revisión menor se corresponde con una mejora incremental a la arquitectura central, posiblemente añadiendo nuevas características.

Las capacidades de cómputo de las GPUs utilizadas en este proyecto fin de carrera son las siguientes:

	Compute Capability
GeForce GTX 770	3.0
GeForce GTX 780	3.5

Tabla 2.2: Capacidades de computo de nuestras GPUs

2.3.4. API

2.3.4.1. CUDA C como extensión del lenguaje C

La meta de la interfaz de programación de CUDA C es proporcionar un camino relativamente simple para que los usuarios familiarizados con la programación en lenguaje C puedan fácilmente escribir trozos de programas que sean ejecutados en la GPU. Por ello:

- Hay un conjunto mínimo de extensiones al lenguaje C que permiten al programador enfocar partes del código fuente para su ejecución en el dispositivo.
- Una librería de *runtime* dividida en:
  - Un componente de host, que corre en el host y proporciona funciones para controlar y acceder a uno o más *devices* desde el *host*.
  - Un componente de dispositivo que corre en el *device* y proporciona funciones específicas para el *device*.

- Un componente común que proporciona tipos de vector propios y un subconjunto de la librería estándar de C que son soportados tanto en el código de host como en el del *device*.

2.3.4.2. Extensiones del lenguaje CUDA C

Las extensiones del lenguaje de programación C están divididas en cuatro:

- Calificadores de tipo de funciones para especificar dónde se ejecuta una función y desde dónde es invocable:
  - **`__device__`:** Se ejecuta en el dispositivo y es llamable sólo desde el *device*.
  - **`__global__`:** Se ejecuta en el dispositivo y es llamable sólo desde el *host*.
  - **`__host__`:** Se ejecuta desde el host y es llamable sólo desde el *host*. Es equivalente a no poner ningún calificador, sin embargo si se combinan los tipos `__host__` y `__device__` el código se compilará para ambos.
- Calificadores de tipo de variables para especificar la localización en memoria en el dispositivo de una variable:
  - **`__device__`:** Variable que reside en el *device*. Ha de ir acompañado de algún otro calificador para especificar su espacio de memoria. En caso contrario, la variable reside en memoria global, tiene el tiempo de vida de una aplicación y es accesible desde todos los hilos del *grid* y desde el *host* a través de la librería de *runtime*.
  - **`__constant__`:** Variable que reside en el espacio de memoria constante, tiene el tiempo de vida de una aplicación y es accesible desde todos los hilos dentro del gridy desde el host a través de la librería de *runtime*.
  - **`__shared__`:** Variable que reside en el espacio de memoria compartida de un bloque de hilos, tiene el tiempo de vida del bloque y es sólo accesible por los hilos del bloque.
- Configuración de ejecución: Cualquier llamada a una función `__global__` debe especificar la configuración de ejecución, que define la dimensión del *grid* y los bloques que serán usados para ejecutar la función en el *device*. Se especifica insertando una expresión del tipo **`<<< Dg, Db, Ns, S >>>`** entre el nombre de la función y la lista de argumentos, donde:
  - **`Dg`** es de tipo dim3 (ver siguiente sección) y especifica la dimensión y tamaño del grid, tal que Dg.x * Dg.y es igual al número de bloques que son lanzados; Dg.z no se usa ya que el *grid* tiene dos dimensiones.

- **Db** es de tipo dim3 (ver siguiente sección) y especifica las dimensiones y tamaño de cada bloque, tal que Dg.x * Dg.y * Db.z es igual al número de hilos por bloque.
- **Ns** es de tipo size_t y especifica el número de bytes en memoria compartida que es reservado dinámicamente por bloque; esta llamada además de la memoria compartida reservada estáticamente; esta memoria reservada dinámicamente es usada por cualquier variable declarada como un array externo; Ns es un argumento opcional cuyo valor por defecto es 0.
- **S** es de tipo cudaStream_t y especifica el *stream* asociado; S es un argumento opcional cuyo valor por defecto es 0.

- Cuatro variables auto-generadas que especifican las dimensiones de *grid* y bloque y los índices de bloque e hilo dentro del kernel:
  - **gridDim:** variable de tipo dim3 (ver siguiente sección) que contiene las dimensiones del *grid*.
  - **blockIdx:** variable d tipo uint3 (ver siguiente sección) que contiene el índice de bloque dentro del *grid*.
  - **blockDim:** variable de tipo dim3 (ver siguiente sección) que contiene las dimensiones del bloque.
  - **threadIdx:** variable de tipo uint3 (ver siguiente sección) que contiene el índice de hilo dentro del bloque.

Cada archivo fuente que contenga estas extensiones debe ser compilado con el compilador de CUDA C nvcc. Además, cada una de estas extensiones viene con algunas restricciones (NVIDIA Corp., 2009).

2.3.4.3 Tipos de vectores propios de CUDA C

Están derivados de los tipos enteros y de punto-flotante básicos. Son estructuras en las que las primeras, segundas, terceras y cuartas componentes son accesibles a través de los campos x, y, z y w respectivamente.

Los tipos básicos son:

- char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4
- short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4
- int1, uint1, int2, uint2, int3, uint3, int4, uint4
- long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4
- float1, float2, float3, float4

Donde u significa **unsigned** (NVIDIA Corp., 2015b).

2.3.5. Optimización del código

2.3.5.1. Ocupancia

La ocupancia indica cuántos de los procesadores de un multiprocesador se encuentran activos. Cuantos menos hilos suspendidos existan, mayor efectividad tendrá el programa. Sin embargo, una ocupancia mayor no implica necesariamente mayor rendimiento. Si la rapidez de ejecución de un kernel no está limitada por el ancho de banda, incrementar la ocupancia no incrementa el rendimiento necesariamente. Si una invocación de un kernel está ya corriendo al menos en un bloque de hilos por multiprocesador en la GPU, y sufre cuello de botella por computación y no por accesos a memoria global, entonces incrementar la ocupancia puede no tener efecto. De hecho, hacer cambios sólo para incrementar la ocupancia puede tener otros efectos, como instrucciones adicionales, código divergente, etc. Como con cualquier optimización, se debería experimentar para ver cómo los cambios afectan el tiempo de ejecución de un kernel. Para aplicaciones con ancho de banda vinculado, por otra parte, incrementar la ocupancia puede ayudar a ocultar mejor la latencia de accesos a memoria, y por lo tanto a mejorar el rendimiento (NVIDIA Corp., 2009).

2.3.5.2. Transferencias de datos entre host y device

El ancho de banda entre el *device* y la memoria del *device* es mucho mayor que el ancho de banda entre la memoria del *device* y la memoria del *host*. Por lo tanto se deberían minimizar las transferencias de datos entre el *hos*t y el *device*, por ejemplo pasando código del *host* al *device* o agrupando muchas transferencias pequeñas en una mayor (NVIDIA Corp., 2009c).

2.3.6. Operaciones en paralelo

La reducción de un vector es una importante operación primitiva de datos en paralelo muy común en numerosos algoritmos. Además, es un muy buen ejemplo de estrategias de optimización a seguir, y tiene especial relevancia en el código de nuestro trabajo ya que se realiza una operación de producto escalar entre dos vectores.

El cálculo del producto escalar de dos vectores se realiza en dos fases: multiplicación de componente a componente de los vectores; y suma de los elementos del vector que almacena los resultados de los productos, esta es la reducción. La Figura 2.18 muestra una

aproximación de la operación en forma de árbol realizada por un bloque de hilos (Harris, 2007).

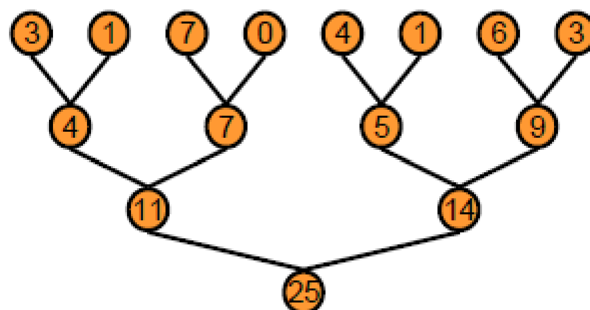


Figura 2.18: Esquema de una reducción

Al implementar una reducción en un kernel, el tamaño de los vectores es relativamente grande, se han de usar varios bloques, cada uno de los cuales resolverá una porción del vector.

Además, el algoritmo de reducción no tiene por qué limitarse a sumar los elementos de un vector, también puede usarse para paralelizar otro tipo de operaciones con las componentes como buscar el valor máximo de un array, por ejemplo. Sin embargo, este algoritmo presenta una serie de restricciones a tener en cuenta en su ejecución:

- Debido a la continua división entre dos del número de hilos que ejecutan la operación, el tamaño del bloque de hilos tiene que ser una potencia de dos.
- No se puede aplicar una reducción a un vector con un número de elementos menor al número de hilos que ejecutan la reducción. Esto se debe al punto de sincronización existente al final de cada iteración, que como ya se ha dicho, debe realizarse en una parte del código común a todos los hilos del bloque. Esta restricción puede solventarse rellenando con ceros el vector hasta ajustar el tamaño.

2.3.6.1. Sincronización global

Cuando todos los bloques produjeran su resultado, un punto de sincronización global permitiría continuar la operación recursivamente. Sin embargo, CUDA C no tiene sincronización global. La solución por lo tanto es descomponer la operación en varias llamadas a kernel, ya que el lanzamiento de un kernel sirve como punto de sincronización global. En el caso de las reducciones, el código de las distintas llamadas es el mismo, por lo que se puede invocar el kernel recursivamente.

2.3.6.2. Accesos coalescentes a memoria global del device

La coalescencia es un acceso a memoria global coordinado, consiguiendo mayor rendimiento, ejecutado por un *warp*. El k-ésimo hilo en un *warp* debe acceder al k-ésimo elemento en un bloque. Aunque no es necesario que participen todos los hilos: véase las figuras Figura 2.19 y Figura 2.20 (Tejero de Pablos, 2009).

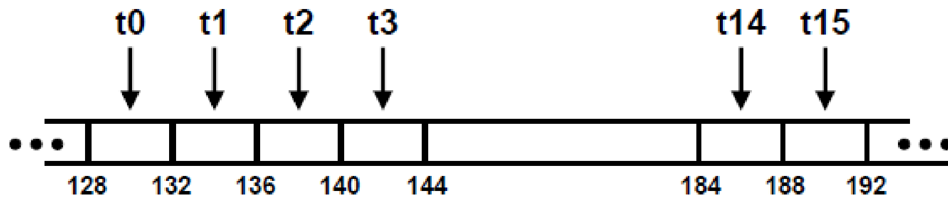


Figura 2.19: Lectura/escritura coalescente con todos los hilos

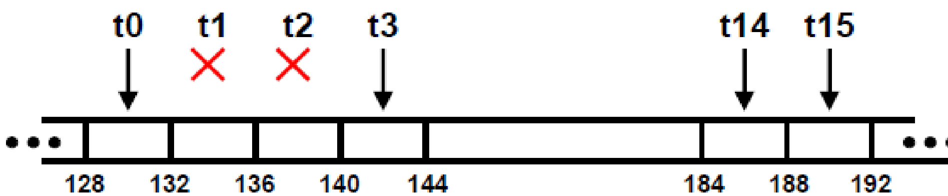


Figura 2.20: Lectura/escritura coalescente con todos los hilos

Un acceso no es coalescente si, por ejemplo, los hilos realizan un acceso permutado, Figura 2.21, o la dirección de inicio no está alineada, Figura 2.22 (Tejero de Pablos, 2009).

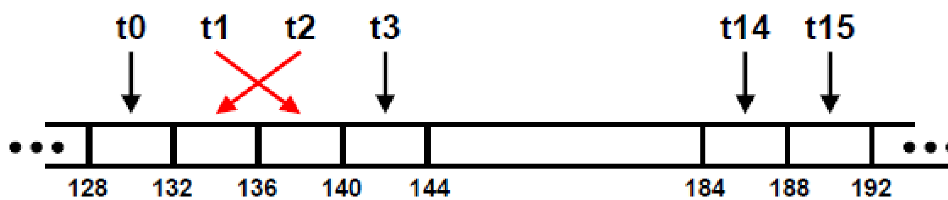


Figura 2.21: Accesos a memoria permutados por los hilos

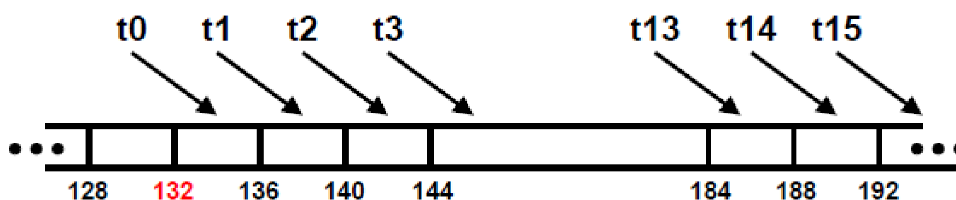


Figura 2.22: Dirección de inicio no alineada con la lectura/escritura de los hilos

2.3.6.3. Conflictos de bancos de memoria compartida

La memoria compartida para un kernel se distribuye en 16 bancos de memoria que se comparten entre los hilos que se ejecutan simultáneamente. La forma de acceder de los hilos a los elementos del vector utilizando direccionamiento no contiguo, véase Figura 2.23 (Tejero de Pablos, 2009), provoca este tipo de conflictos, ya que un mismo hilo accede a posiciones de memoria contenidas dentro del mismo banco (Harris, 2007).

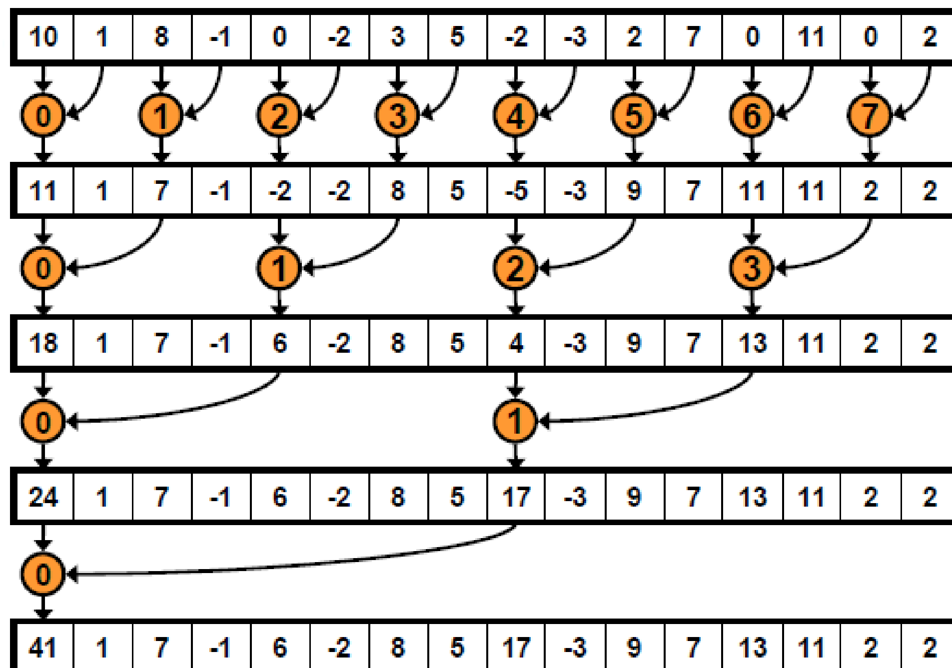


Figura 2.23: Reducción con conflictos de bancos

La solución se encuentra entonces en que distintos hilos accedan a posiciones consecutivas de memoria, como se muestra en la Figura 3-24 (Tejero de Pablos, 2009).

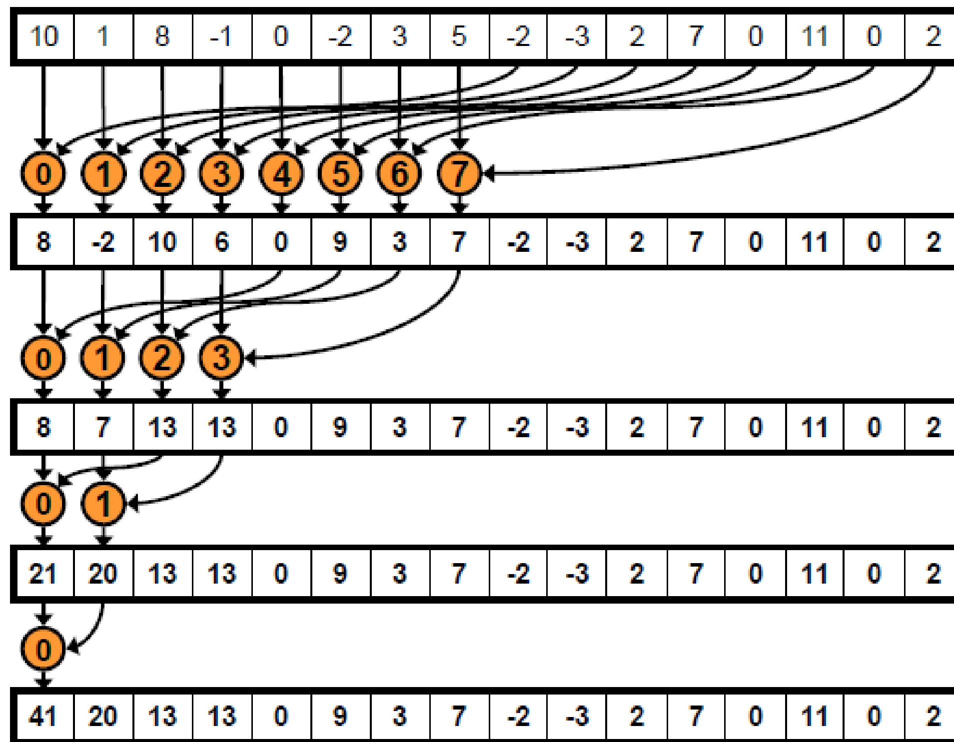


Figura 2.24: Reducción sin conflictos de bancos

Un algoritmo de reducción de un vector puede mejorarse aún más con nuevas modificaciones, como realizando operaciones durante la carga de los vectores o "desenrollando" el bucle para no realizar operaciones sobre elementos que no nos interesan. Como el objetivo de este trabajo no es optimizar la reducción de un vector, sino hacer un programa genérico que se ejecute en multi-GPU, las mejoras más avanzadas de la reducción no han sido implementadas.

2.3.7. CUDA Streams

Un *stream* de CUDA es una secuencia de operaciones que se ejecuta en el *device* en el orden que se especifica en el código. Mientras que en las operaciones de un mismo *stream* está garantizado que se ejecutarán en el orden descrito, las operaciones de *streams* diferentes se pueden intercalar, y si es posible, pueden ejecutarse simultáneamente.

2.3.7.1. El stream por defecto (default stream)

Todas las operaciones del *device* (ejecuciones de los kernels y transferencias de datos) en CUDA se ejecutan en un *stream*. Cuando ningún *stream* se especifica, se utiliza el *default*

stream. La característica especial de este *stream*, que le hace único, es que es un *stream* síncrono, esto es, que ninguna operación en el *stream* por defecto comenzará hasta que todas las operaciones previas de cualquier *stream* del *device* se hayan completado, y que ninguna otra operación de cualquier *stream* comenzará hasta que no se complete la operación que corresponda al default *stream* (Harris, 2012).

2.3.7.2. Streams (Non-default streams)

Para utilizar *streams* diferentes al *stream* por defecto es necesario declararlos y crearlos antes de utilizarlo, y una vez terminado su uso se deben destruir. Además, para las copias de datos se deberá utilizar una función específica y al lanzar el kernel se le especificará el *stream* que se utilice (ver ejemplo de aprendizaje nº 6).

Como todas las operaciones en *streams* diferentes al default *stream* son no bloqueantes con respecto al código que se ejecuta en el host, en algunos casos será necesaria la sincronización y se utilizarán diferentes funciones que proporcionan diferentes grados de sincronismo.

Un aspecto de primordial importancia cuando trabajamos con *streams* es el orden en que planeamos las transferencias de datos y la ejecución de los kernels de los diferentes *streams*, ya que de ello dependerá que se obtenga o no una implementación eficiente en la que se solapen las distintas operaciones. La diferencia de tiempo que se tarda en procesar los mismo datos dependiendo de cómo se planifique se puede ver claramente comparando la versión 1 de la versión 2 de la Figura 2.25. En las GPUs más modernas con una capacidad de cómputo igual o superior a 3.0 se dispone de dos planificadores de copias, uno que se encarga de las que se realizan del *host* al *device* y otro a la inversa, lo que permite que también se puedan solapar diferentes copias de datos (versión 3) (Harris, 2012).

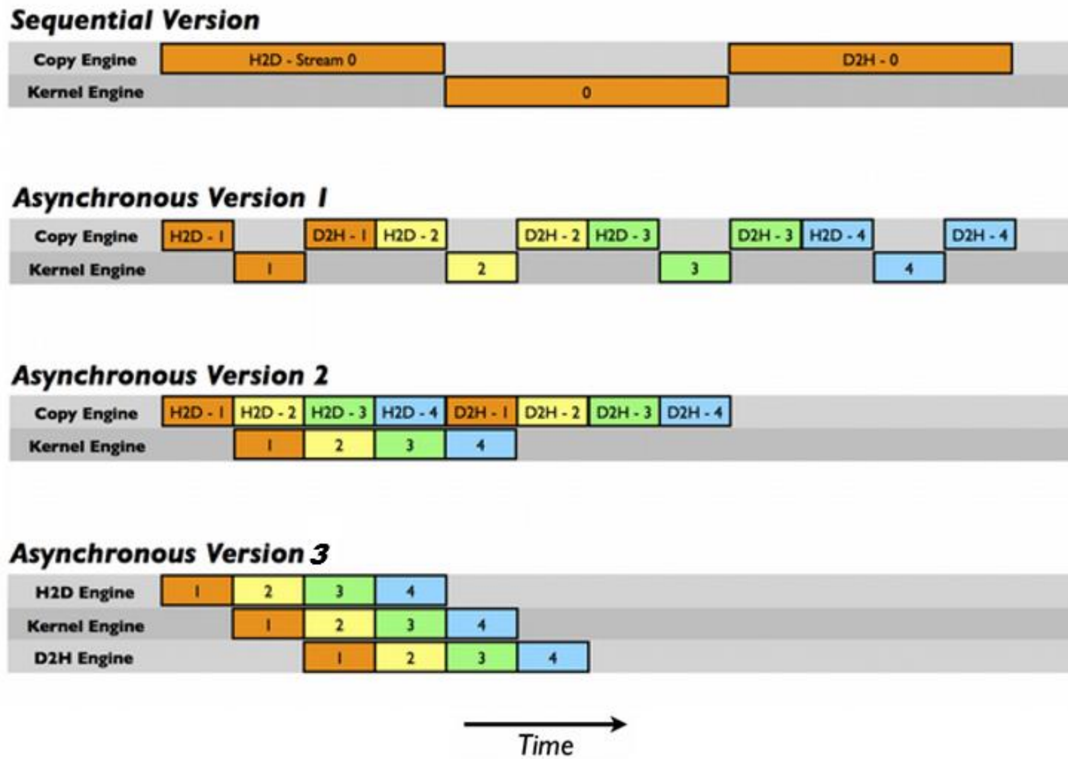


Figura 2.25: Diferentes planificaciones de streams para las mismas operaciones

2.3.8. Multi-GPU

Las dos principales razones por las que tener interés en utilizar multi-GPU son que se consigue una mayor aceleración de los cálculos y/o que la memoria de una GPU no sea suficiente para la cantidad de datos en cierto trabajo (Micikevicius, 2011). Trabajando con varias GPUs puede ser necesaria la comunicación entre ambas, y en este apartado se hablará sobre las dos diferentes alternativas en las que varias GPUs pueden ser instaladas en un sistema y las implicaciones para los desarrolladores de CUDA.

La primera opción consiste montar cada tarjeta gráfica con una única GPU en su interior en una ranura PCI Express. En el 2004 aproximadamente NVIDIA introdujo la tecnología SLI (*Scalable Link Interface*) que permitía que múltiples GPUs trabajaran en paralelo para obtener un mejor rendimiento en las tareas gráficas. Estas GPUs se montaban sobre distintas ranuras PCI Express de la placa base, y los drivers de NVIDIA se encargaba de hacer que estas GPUs aparecieran como una única al sistema (Wilt, 2013).

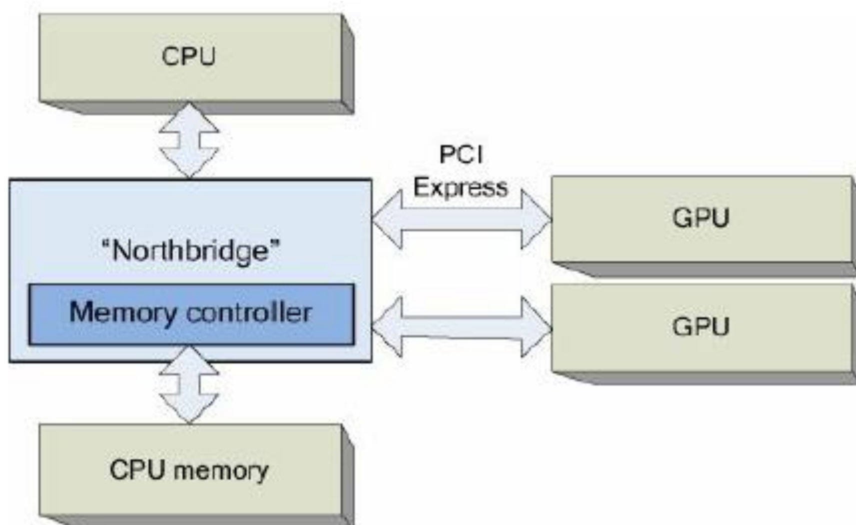


Figura 2.26: GPUs en varias ranuras (slots)

La otra posibilidad para disponer de un sistema multi-GPU es disponer de una tarjeta gráfica que contiene en su interior más de una GPUS. Algunos ejemplos de estas tarjetas son la GeForce 9800GX2 (dual-G92) y la GeForce GTX 295 (dual-GT200), entre otros. Lo único que comparten estas tarjetas es el *bridge chip* que permite a ambas GPUs comunicarse con la CPU mediante una única ranura PCI Express. Las GPUs que forman la tarjeta no comparten memoria, cada una de ella tiene su propia memoria, aunque existen formas de comunicarse entre las dos GPUs, e incluso para las GPU de arquitectura Fermi y posteriores una GPU podrá mapear direcciones de memoria de la otra GPU en su espacio de direcciones global (Wilt, 2013).

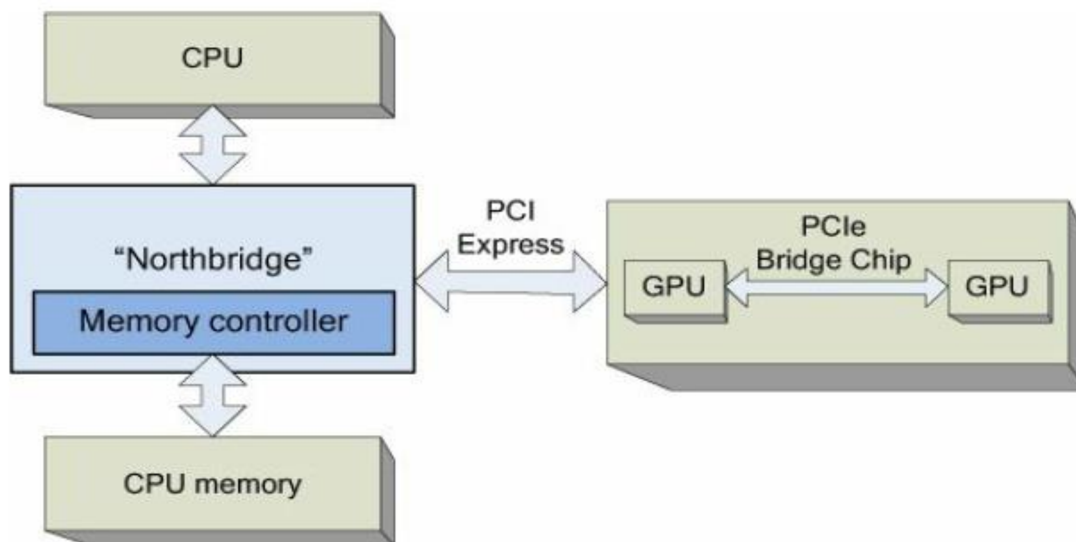


Figura 2.27: varias GPUs en una única tarjeta

2.4. CUDA en MATLAB

MATLAB es una herramienta de simulación ampliamente utilizada para la realización rápida de prototipos y el desarrollo de algoritmos. Muchos laboratorios e instituciones de investigación utilizan MATLAB en sus trabajos, y constantemente incrementan la demanda de rapidez en la ejecución de los códigos de MATLAB. Esto, junto a la representación vectorial o matricial de los datos que utiliza MATLAB, hace que el uso de la GPU se presente como una magnifica alternativa para la aceleración de los cálculos (W. Suh y Kim, 2014).

El empleo de la GPU para aumentar la velocidad de cálculo de los algoritmos en MATLAB se puede hacer de dos maneras diferentes. Una de ellas es haciendo uso de la herramienta de MATLAB de cálculo paralelo (*Parallel Computing Toolbox*), que permite solucionar problemas de alta carga computacional así como trabajar con grandes cantidades de datos utilizando para ello procesadores con varios núcleos (*multicore processors*), GPUs y clusters de ordenadores. La otra opción es utilizar lo que se conoce con el nombre de ficheros mex, con los cuales es posible realizar llamadas a tus propias subrutinas escritas en C como si de funciones de MATLAB se tratase.

Algunas de las ventajas que proporciona esta segunda opción, que es la que se utiliza en este trabajo, son: una mayor flexibilidad, ya que somos nosotros los que nos encargamos de programarlo y así podremos obtener mayores mejoras en la aceleración de nuestros cálculos; la superación de las limitaciones de la herramienta de MATLAB para la optimización de cálculos en la GPU; y el ahorro del coste extra que supone adquirir una licencia para el *Parallel Computing Toolbox* (W. Suh y Kim, 2014).

2.4.1. Archivos c-mex

Los archivos c-mex proceden de ejecutables MATLAB (*MATLAB Executable*, *MEX files*), y son funciones que se pueden llamar desde MATLAB escritas en C/C++. El sentido de estos archivos está en que con ellos es posible proporcionar velocidades de ejecución más alta y evitar los cuellos de botella en las aplicaciones (W. Suh y Kim, 2014).

Para crear un archivo mex se necesita:

- Tener un código fuente escrito en C o C++.
- Un compilador que sea soportado por MATLAB. En este trabajo se ha utilizado GCC.

- La API de C/C++ para manejar la estructura de datos de MATLAB (mxArray) (C/C++ Matrix Library API) y la librería MEX para realizar operaciones del entorno de MATLAB desde el código en C/C++ (MathWorks, 2015).

2.5. Ejemplos de aprendizaje

En este apartado se van a presentar una serie de ejemplos de códigos en CUDAD C realizados durante mi aprendizaje y que pueden servir como buen punto de partida para futuros estudiantes que se quieran introducir en este lenguaje. La dificultad de los ejemplos va aumentando progresivamente a medida que se avanza sobre los mismos, partiendo desde un ejemplo muy sencillo hasta el último en el que ya se puede apreciar una cierta complejidad.

2.5.1. Ejemplo 1: Introducción a CUDA C

Como no podía ser de otra manera, el primer ejemplo que vamos a tratar aquí es el "Hola mundo", pero con una pequeña variación, ya que vamos a imprimir nuestro saludo en pantalla desde el dispositivo (la GPU).

Para que este primer ejemplo funcione es necesario que la capacidad de cómputo de nuestra GPU sea 2.0 o superior (en nuestro caso estamos trabajando con una GPU de capacidad 3.5) para poder imprimir en pantalla desde el kernel. También debemos asegurarnos de que el kernel ha terminado su ejecución antes de dar por terminada la función `main` (`cudaDeviceSynchronize()`).

```
#include <stdio.h>

__global__ void kernel (void){
	printf("Hello world from the device!\n");
}

int main(void){
	kernel <<<1,1>>>();
	cudaDeviceSynchronize();
	return 0;
}
```

Figura 2.28: Código ejemplo "Hola Mundo"

En la Figura 2.28, en la llamada al kernel vemos que lanzamos un único bloque con un único hilo. Si lanzásemos más de un hilo, ya fuese en el mismo bloque o en distintos, el

kernel se ejecutaría tantas veces como hilos lanzados y tendríamos tantos saludos como hilos.

2.5.2. Ejemplo 2: Reservas de memoria

El segundo ejemplo es una simple suma de enteros realizada en el kernel, pero que fuerza a introducir novedades en nuestro código. En este caso realizamos una reserva de memoria en el dispositivo antes de la llamada al kernel para la posición apuntada por el puntero que pasaremos como argumento al kernel y que se encarga de almacenar el resultado. Como es una reserva de memoria en el dispositivo la función a la que llamamos es `cudaMalloc()`, a la que le pasamos como argumento el tamaño de la reserva, y nos devuelve un puntero al espacio de memoria reservado en el dispositivo. Se llama al kernel lanzando un único bloque con un hilo, y pasándole dos argumentos como parámetro y un tercero como referencia, que es el puntero a donde almacenar el resultado. Tras la llamada al kernel se copia el resultado desde el *device* al *host* mediante la función `cudaMemcpy()`, en la que indicamos la dirección destino y origen, el número de bytes copiados, y el tipo de copia (`cudaMemcpyDeviceToHost`).

```
#include <stdio.h>

__global__ void add(int a, int b, int *c){
	*c = a + b;
}

int main(void){
	int c;
	int *dev_c;

	cudaMalloc((void**)&dev_c, sizeof(int));

	add<<<1,1>>>(2, 7, dev_c);

	cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);

	printf("2 + 7 = %d\n", c);

	cudaFree(dev_c);

	return 0;
}
```

Figura 2.29: Código ejemplo reserva de memoria

2.5.3. Ejemplo 3: Pasando argumentos al kernel y ejecución de múltiples hilos

El siguiente paso, después del ejemplo anterior visto, sería pasar al kernel argumentos de entrada diferentes a una constante, pero en esta ocasión se van a dar dos pasos de una sola vez y vamos a ver también la ejecución en paralelo de múltiples hilos. Para ello se va a realizar la suma de dos vectores elemento por elemento, y se almacenará el resultado en un tercer vector.

Se comienza declarando los vectores en el host, reservando memoria para los mismos e inicializándolos. A continuación se declaran los correspondientes vectores homólogos para el dispositivo, y se reserva también memoria para ellos, esta vez en la memoria de la GPU, y entonces se pasa a realizar la copia del host al dispositivo de los vectores que queremos sumar. En este punto ya casi estamos en condiciones de lanzar el kernel, pero antes tenemos que determinar el número de bloques y de hilos necesarios. Para ellos es una práctica habitual fijar el número de hilos por bloque (`threadsPerBlock`) que se desean para optimizar el código, y en función del número de hilos por bloque calcular el número de bloques (`blocksPerGrid`) lanzados siguiendo la fórmula:

```
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
```

De este modo al lanzar el kernel lo hacemos con el mínimo número de recursos suficientes que satisfacen nuestro objetivo, y así evitamos sobredimensionar los recursos. También es cierto que puede ocurrir que si el número de hilos por bloque no es divisor del número de elementos de los vectores, en el último bloque lanzado habrá hilos innecesarios que no realizarán ninguna operación.

Una vez determinados el número de hilos y bloques necesarios, se lanza el kernel pasándole como argumentos los tres vectores (los dos a sumar y donde almacenaremos el resultado) y el tamaño de los mismos.

En el kernel asignamos a cada hilo una posición de los vectores que se encarga de sumar:

```
int tid = blockDim.x * blockIdx.x + threadIdx.x;
if (tid < numElements){
    C[tid] = A[tid] + B[tid];
}
```

Después de la llamada al kernel copiamos el resultado almacenado en la memoria de la GPU a la de la CPU, comprobamos que el resultado es correcto, y liberamos la memoria reservada tanto en el *device* como en el *host*.

Como anexo a este ejemplo decir que es una práctica habitual y muy aconsejable realizar una comprobación de errores en cada llamada a una función de CUDA C, por ello se suele definir una macro, en estos ejemplos `HANDLE_ERROR()`, que realiza esta

comprobación de errores e informa, si sucede uno, de qué error se trata y dónde se localiza, a la vez que se sale de la aplicación.

2.5.4. Ejemplo 4: Eventos de CUDA

El cuarto ejemplo introduce el concepto de eventos, que permiten medir el rendimiento de los programas. Cuando se realizan cambios en el código, cuando se utilizan unas u otras funciones, o cuando se hacen las operaciones de diferentes maneras siempre cabe la pregunta de ¿qué versión tarda menos en realizar las operaciones? Para medir tiempos de ejecución se puede utilizar la CPU o los temporizadores del sistema operativo, pero estos incluirán latencias y pequeñas variaciones procedentes de diferentes fuentes. Además mientras se lanza un kernel en la GPU, es posible que se estén realizando otros cálculos en el lado de la CPU. De tal manera que la forma apropiada para medir tiempos es utilizar temporizadores del sistema operativo para operaciones ejecutadas en la CPU, y la API de eventos de CUDA para medir operaciones en la GPU.

Un evento en CUDA es básicamente una marca de tiempo grabada por la GPU en un momento específico determinado por el usuario. Así pues, si se quiere medir el tiempo de un bloque de código, se deberán crear los eventos de inicio (*start*) y de parada (*stop*), a continuación se grabará el tiempo en el evento de inicio, se realizarán las operaciones que se desean temporizar, se grabará el tiempo en el evento de parada, se esperará a que se hayan completado las operaciones previas a la marca del evento de parada, y se medirá el tiempo transcurrido entre ambos eventos (Sander y Kandrot, 2010).

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
   cudaEventRecord(start,0);

// Do some work on the GPU

cudaEventRecord(stop,0);
cudaEventSynchronize(stop);


// Print the elapsed time between the two previously recorded events
float elapsedTime;
cudaEventElapsedTime(&elapsedTime,start,stop);
printf("Elapsed time (GPU working time): %.6f ms\n", elapsedTime);
```

Figura 2.30: Código ejemplo eventos

2.5.5. Ejemplo 5: Memoria compartida

En este ejemplo se va a dar un importante paso y se va a introducir el concepto de memoria compartida. Para hacer uso de la memoria compartida y de la cooperación entre hilos que esta permite se va a realizar el producto escalar de dos vectores, en el que es necesario realizar una reducción del vector que almacena los productos elemento por elemento para sumarlos todos en un único valor.

La memoria compartida (*shared memory*) es una memoria on-chip, por lo que es mucho más rápida que la memoria local y global, de echo puede llegar a reducir 100 veces la latencia en comparación con la memoria global cuando no se producen conflictos de bancos entre hilos (un *bank conflict* ocurre cuando hilos del mimo *warp* solicitan diferentes valores del mismo banco en una única petición. No ocurren *bank conflits* cuando todos los hilos de los diferentes *wrap* acceden a los mimas posiciones de cada banco, el mimo patrón). La memoria compartida está asignada por cada bloque de hilos, y por tanto todos los hilos de un bloque tienen acceso a la misma memoria compartida, lo que les permite, junto con la sincronización de hilos, desempeñar tareas paralelas de cooperación (Harris, 2013).

Cuando realizamos operaciones en las que compartimos los datos entre diferentes hilos, es necesario ser cuidadoso de evitar condiciones de carrera, porque aunque todos los hilos en un bloque se ejecutan lógicamente en paralelo, no todos se pueden ejecutar físicamente en el mimo momento. Para evitar resultados inesperados se debe sincronizar los hilos, y para ello utilizar la primitiva `__syncthreads()`, que fuerza a que la ejecución no se pueda continuar hasta que todos los hilos del bloque hayan ejecutado `__syncthreads()`. Así pues, se tendrá que llamar a dicha función después de almacenar cualquier dato en la memoria compartida y antes de que algún hilo lea esta memoria (Harris, 2013).

En CUDA C la memoria compartida se puede declarar de manera estática, cuando es conocido el tamaño del array de memoria compartida en el tiempo de compilación y se declara explícitamente; o como una variable externa, cuando no se conoce el tamaño del array necesario, y entonces se determina el tamaño de memoria por bloques a través de un argumento de configuración en el momento del lanzamiento del kernel.

```
__shared__ float cache[256];
extern __shared__ float cache[];
```

[...]

```
dot<<<blocksPerGrid, threadsPerBlock, threadsPerBlock*sizeof(float)>>>();
```

La cooperación entre hilos se realiza en el contexto de reducción del vector que almacena los resultados de los productos elemento por elemento. El primer paso a seguir para realizar la reducción de un vector es escribir los valores de nuestro vector en la memoria compartida y realizar una llamada a `__syncthreads()` para asegurarnos que antes de continuar todos los hilos han escrito el dato correspondiente en la memoria. En este punto la idea general es que cada hilo sume dos elementos del vector almacenado en `cache[]` y lo vuelva a guardar ahí, para ello se realiza la suma, elemento con elemento, de una mitad con la otra, reduciendo así el tamaño del vector a la mitad. Realizando esta operación sobre la mitad resultante sucesivamente, después de log2(threadsPerBlock) veces, conseguimos sumar todos los elementos y tener el resultado en un único valor (Sander y Kandrot, 2010).

Después de realizar el bucle `while`, cada bloque tiene almacenado en la primera posición del array `cache[]` el resultado de la suma de los elementos correspondientes, entonces guardamos este único valor en la memoria global y finalizamos nuestro kernel (Sander y Kandrot, 2010).

```
    cache[threadIdx.x] = temp;

    __syncthreads();

    // cache[] vector reduction
    int n = blockDim.x/2;
    while (n !=0){
       if (threadIdx.x < n)
              cache[threadIdx.x] += cache[threadIdx.x + n];
       __syncthreads();
       n /= 2;
    }

    if (threadIdx.x == 0)
       C[blockIdx.x] = cache[0];
```

Figura 2.31: Código ejemplo memoria compartida

2.5.6. Ejemplo 6: Streams

En este ejemplo se trabaja con *streams*, que permitirán dar un paso más en la paralelización de algoritmos. Un *stream* en CUDA C representa una cola de operaciones de la GPU que esta ejecutará en el orden especificado. Las operaciones que se pueden añadir a un *stream*, que se ejecutarán en el orden en que se añaden, son lanzamientos de kernels, copias en memoria, y eventos de inicio y parada (Sander y Kandrot, 2010). Cuando se trabaja con más de un *stream* puede suceder que las distintas operaciones programadas para cada *stream* puedan ejecutarse intercaladas o incluso simultáneamente y se solapen, obteniéndose así nuevos niveles de concurrencia.

Para lograr el solapamiento de ejecuciones del kernel con copias de memoria es necesario que el dispositivo tenga una capacidad de cálculo de 1.1 o superior; que estas operaciones pertenezcan a *streams* diferentes, y distintos a su vez del *stream* por defecto; y que la memoria del host involucrada en la copia no sea paginada (page-locked o *pinned memory*) (Harris, 2012). Así pues si se van a utilizar *streams* y se desea que la memoria permanezca fija en memoria física, la reserva de memoria en la CPU se realiza con la función

```
cudaHostAlloc((void**)&h_A, size, cudaHostAllocDefault)
```

En las copias de memoria, para que sean asíncronas y hacer uso de streams se utilizará

```
cudaMemcpyAsync(d_A, h_A, , size, cudaMemcpyDeviceToHost, stream1);
```

En el lanzamiento del kernel se añade un cuarto parámetro en el que se indica el *stream* al que se añade esta ejecución:

```
vectorAdd<<<blocksPerGrid, threadsPerBlock, 0, stream1>>>(d_A, d_B, d_C,
numElements);
```

Y también si se quiere garantizar que la GPU ha realizado todas las operaciones correspondientes en un determinado momento, se realiza una llamada a la función de sincronización similar a la vista en el punto anterior para los hilos:

```
cudaStreamSynchronize(stream1)
```

2.5.7. Ejemplo 7: multi-GPU

En este ejemplo se tratará la ejecución multi-GPU, con la cual se conseguirá un aumento de la velocidad de ejecución de nuestros programas y/o trabajar con datos que excedan la capacidad de memoria de una única GPU (Micikevicius, 2011). Lo primero que se ha de saber de la ejecución multi-GPU es que las distintas GPUs de nuestro sistema pueden ser controladas por un único hilo de la CPU, por varios hilos de la CPU pertenecientes al mismo proceso, o por múltiples procesos de la CPU (caso este que no explicaremos). Estas distintas formas de controlar una GPU posibilitan diferentes alternativas a la hora de escribir código.

La primera opción de dirigir varias GPUs desde un único hilo de la CPU (`p-thread`) se basa en el empleo de la función `cudaSetDevice()` combinada con otras funciones (kernels, copias de memoria) llamadas de manera asíncronas. De esta manera lo que se hace es crear una cola de tareas para una GPU, cambiar a otra GPU y encolar otras tantas tareas a esta. Cuando se utiliza esta técnica hay que ser cuidadoso y tener presente el

dispositivo que está elegido en cada momento que realizamos una llamada a otra función, así como el *stream* en que colocamos dicha tarea.

La segunda opción, la de crear nuevos hilos por cada GPU añadida, es la que se muestra en este ejemplo en concreto. La operación a realizar es el producto escalar de dos vectores, y para realizarlo en un tiempo menor, se dividen los vectores a la mitad y se asigna cada mitad a una GPU. En la función `main()` del ejemplo se realizan las funciones habituales de reserva de memoria, iniciación de los vectores, etc., e iniciamos también una estructura creada para pasarle argumentos a la función que realizará la llamada al kernel en cada GPU. A está función se la llama cuando creamos un hilo que asignamos a una GPU, y le pasamos una mitad de los datos, y a continuación la llamamos también desde el propio hilo dado del proceso con la mitad restante para que se ejecute en la otra GPU.

```
err = pthread_create(&(thread), NULL, routine, &(data[0]));
routine(&(data[1]));
```

En esta función `routine()` se selecciona la GPU en la que se va a trabajar, se realizan reservas de memoria en el lado del dispositivo, se copian los datos, se lanza el kernel, se copian los datos en el *host*, se termina de hacer la reducción de la mitad del vector, y se almacena este resultado en una variable de la estructura que pasamos como argumento cuando llamamos a la función. Cuando volvemos al `main()`, tras la ejecución de las tareas en ambas GPUs, solo queda sumar los valores devueltos.

En resumen, las claves para trabajar con esta segunda opción son: la declaración de una estructura que nos sirve para comunicar datos con la función que ejecuta cada GPU, y esta propia función (`routine()` en nuestro caso) en la que se realizan las operaciones correspondientes de cada GPU.

Capítulo 3

Descripción del código

En este apartado se describe el código del programa. Una vez que ya se han explicado los conceptos básicos de la programación en CUDA C, pasamos a estudiar un programa complejo donde se ven actuando juntos muchos de los aspectos previamente comentados.

3.1. Introducción

La operación principal que se realiza en el programa es el cálculo del producto escalar entre dos vectores, pero más allá de este simple cálculo uno de los objetivos principales de este trabajo es proporcionar un esqueleto de programa que sirva para futuros trabajos en los que se quiera hacer uso de GPGPU. Por esta razón, en el código se trata de separar en distintos bloques lógicos las partes comunes a cualquier programa que haga uso de GPGPU, para que de esta manera con realizar el menor número de cambios pertinentes en funciones tales como la reserva de memoria, copia de datos o en el algoritmo del kernel el resto del código se pueda reutilizar.

Las diferentes partes en las que podemos dividir el programa son las siguientes:

1. Recogida y comprobación de los argumentos de entrada del programa.
2. Inicialización de los temporizadores que nos servirán para hacernos una idea del rendimiento de nuestro programa.
3. Inicialización de las GPUs.
4. Hacer las reservas de memoria necesarias.
5. Realizar la copia de los datos en el dispositivo.
6. Procesar los datos.
7. Copiar los resultados a la memoria del *host*.
8. Mostrar los resultados.
9. Liberar los espacios de memoria reservados tanto el *host* como en *device*.
10. Mostrar temporizadores y eliminarlos.

Una de las características más destacadas de este programa es el empleo configurable de *streams* y varias GPUs, que se pasa a describir a continuación. Aunque la operación que se realiza sobre los vectores de entrada es el producto escalar de los mismos, el código del

programa está pensado para que se pueda realizar cualquier operación que reciba dos vectores como entrada con el mínimo número de cambios. La idea del empleo de GPGPU surge al pensar que estos vectores de entrada serán de muy grandes dimensiones. Si esto es así cabe hacerse la pregunta de cómo se puede sacar el máximo rendimiento al uso de la GPU, para realizar la(s) operación(es) en el menor tiempo posible, y de aquí surge la idea de la división de los vectores de entrada en *chunks* y el uso de *streams*.

Cuando se ejecuta el programa uno de los argumentos de entrada es el número de *chunks* que vamos a utilizar, o lo que es equivalente, el número de trozos en el que se van a dividir los vectores de entrada. De esta manera, en nuestro programa son los distintos *chunk*s los que se procesarán (incluyendo las pertinentes reservas y copias de memoria) y los que cada uno de ellos dará un resultado parcial que después se sumará. Otro argumento de entrada muy importante del programa es el número de *streams* que vamos a utilizar. Este número de *streams* afecta a su vez a cada *chunk*, es decir, que en el momento de procesar un *chunk* este se subdividirá en el número de *streams* especificado.

En el caso de que sea posible, y se haga uso de la programación multi-GPU, argumento también configurable en el momento de ejecución del programa, los *chunks* se irán procesando alternativamente en cada una de las GPUs disponibles.

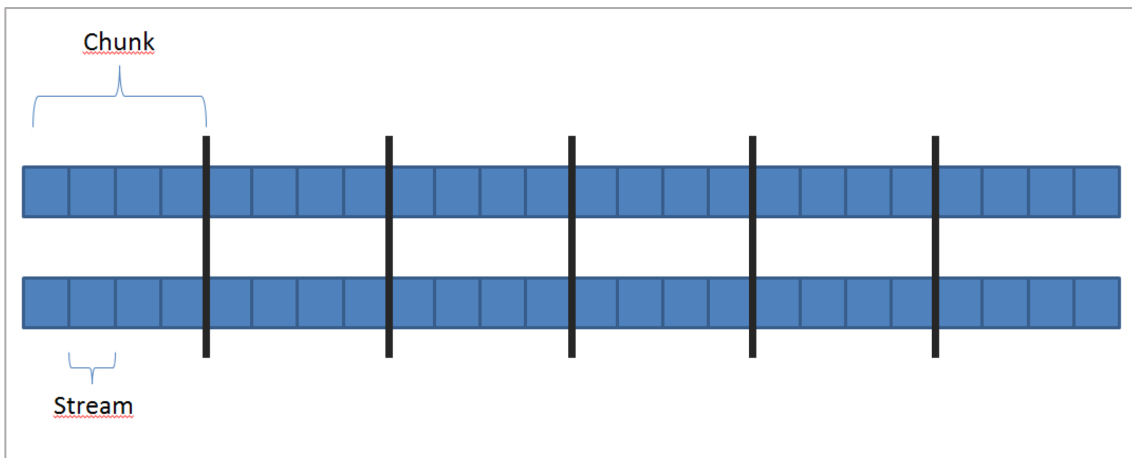


Figura 3.1: División de los vectores en chunks y en streams

Antes de pasar a describir algunas de las funciones de las que consta el programa, es necesario señalar la forma en la que se organizan las diferentes variables del programa, ya que es de vital importancia para la comprensión del código. Todas las variables se articulan entorno a las siguientes cuatro estructuras de datos:

Program: esta es la superestructura que incluye al resto, las que se describen abajo. Además de estas estructuras también se incluyen las variables necesarias para medir los

tiempos de ejecución, y otras variables útiles relacionadas con los chunks, número de elementos del vector, número de GPUs, etc. Se instancia una única variable de este tipo.

GPUacces: es la estructura que contiene los datos y la información de control de la configuración a la que cada GPU debe acceder independientemente, y para ello incluye las dos siguientes estructuras descritas. Se instancian tantas variables de tipo GPUacces como número de GPUs se utilicen.

DataGPUControl: variables de control para las GPUs, principalmente las variables necesarias para la utilización de *streams* y de eventos. El número de instancias de esta variable es igual al número de *chunks* en que dividimos el problema.

DataCPUGPUModel: variables relacionadas con los datos a procesar propiamente dichos, se incluyen los de la parte de la GPU y de la CPU. Al igual que la variable anterior se instancia tantas veces como número de *chunks*.

3.2. Referencia de funciones

El código del programa se encuentra dividido en cuatro ficheros fuente, tres con extensión .cu y otro con extensión .cuh que es el fichero de cabecera. Además de estos cuatro archivos el programa cuenta con uno más de extensión .cpp, que es a partir del cual se crea el fichero mex.

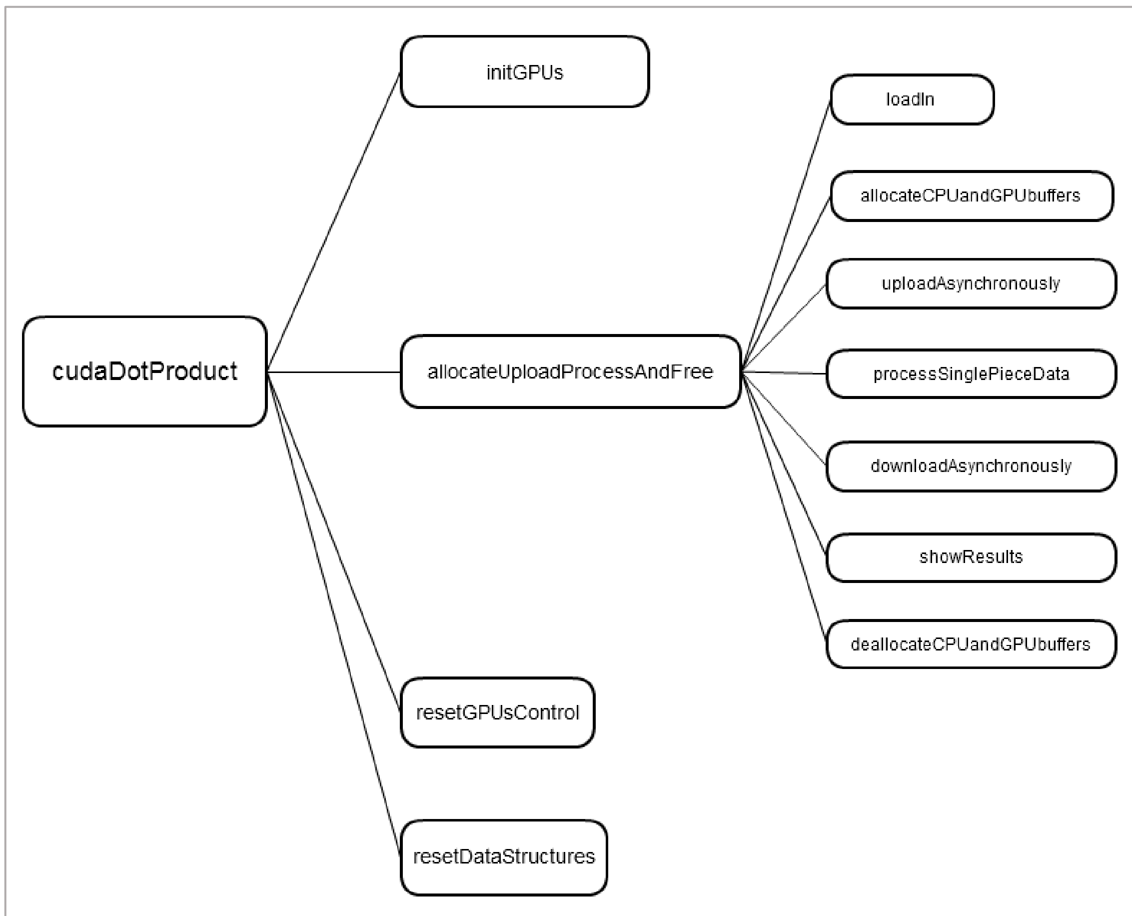


Figura 3.2: Esquema de las funciones del programa

En la Figura XX se muestra un esquema de cómo están anidadas las diferentes funciones del programa, y a continuación se pasa a describir las funciones definidas en cada fichero, comenzando por el fichero cudaDotProduct.cu que es donde está escrito el esqueleto principal del programa.

`void initGPUs (Program* p)`

Función que realiza las tareas de inicialización de las GPUs. Realiza la reserva de memoria para las diferentes estructuras de los datos, y crea los streams, eventos y temporiazadores por cada estructura de datos (por cada chunk).

`void allocateUploadProcessAndFree (Program* p)`

Función principal del código. En esta función se realizan las operaciones de reserva de memoria en la parte de la GPU, copia de los datos de la CPU a la GPU, se lanzan los kernels para procesar los datos, y se copian los resultados de la GPU a la CPU. También se

muestran los resultados al final, y durante toda la función se realizan tareas de temporización.

`void resetGPUsControl (Program* p)`

Función que se encarga de destruir los streams y eventos creados por cada estructura de datos, así como la memoria de los distintos temporizadores. También se resetean las GPUs.

`void resetDataStructures (Program* p)`

Función que libera los espacios de memoria de las estructuras DataGPUControl y DataCPUGPUModel.

`int cudaDotProduct(float* results, float* h_a, float* h_b, int nelements, int nelementschunk, int nchunksjoined, int ngpus, int nstreamsperdata)`

Función principal del programa, desde la cual se llama al resto de funciones, y que sirve de punto de enlace con el archivo de C/C++ que se utiliza para crear el fichero mex. Los argumentos de esta función son los siguientes:

- results: variable en la que se almacenará el resultado de la operación una vez realizada.
- h_a y h_b: vectores sobre los que se realiza la operación
- nelements: número de elementos de los vectores
- nelementschunk: número de elementos de los *chunk*. También podría ser el número de *chunks* en que dividimos los vectores de entrada.
- nchunksjoined: número de *chunks* que juntamos para que se procesen en una sola llamada a una GPU (lo que pasamos a cada GPU para que lo procese se le conoce en el código del programa como data).
- ngpus: número de GPUs que se utilizan.
- nstreamsperdata: número de *streams* que utilizamos para procesar cada data, es decir, cada *chunk* o cada grupo de *chunks* que hayamos unido.

En esta función se crean algunas variables de soporte para el programa, y se configuran algunos aspectos de la ejecución del programa, como el reutilizar o no los buffers reservados en memoria en cada GPU.

Las siguientes funciones se encuentran definidas en el fichero program_logic.cu, y estas funciones son las que más íntimamente relacionadas están con la lógica del programa, es decir, que dependen más de la operación que realicemos. Estas funciones son las que se llaman en cada procesamiento de un *chunk* (data en el programa).

`void allocateCPUandGPUbuffers(DataCPUGPUModel* data, DataCPUGPUModel* dataprevious, DataGPUControl* datacontrol, bool allocatebuffers)`

Función en la que se realizan las reservas de memoria necesarias en el lado de la CPU o la GPU. Cuando está activa la opción de reutilizar los buffers, en esta función asignamos los espacios de memoria ya previamente reservados en el procesado del data anterior para el *data* actual.

`void deallocateCPUandGPUbuffers(DataCPUGPUModel* data, int freeCPU)`

Función en la que liberamos los espacios de memoria de la CPU y la GPU.

`void uploadAsynchronously(DataCPUGPUModel* data, DataGPUControl *datacontrol, int recorrerVec, int gpu, int datachunk)`

Función en la que se realizan las copias de los datos a procesar de la memoria del host a la memoria del *device*.

`void downloadAsynchronously(DataCPUGPUModel* data, DataGPUControl *datacontrol, Program *p, int recorrerVec)`

Función en la que se realizan las copias de los resultados de la memoria del *device* a la memoria del *host*.

`void processSinglePieceData(DataCPUGPUModel* data, DataGPUControl *datacontrol)`

Función en la que se realizan las llamadas a los kernels para el procesamiento de cada *chunk*.

`void showResults(DataCPUGPUModel* data, DataGPUControl *datacontrol, Program *p)`

Función en la que se termina de realizar la reducción del vector de resultados, se suman todas las reducciones parciales obtenidas de los cálculos de la GPU. También se calcula el resultado en la CPU y se muestran ambos resultados por consola.

`void lastData(DataCPUGPUModel* data, DataGPUControl *datacontrol, Program *p, int gpu, int datachunk)`

Función a la que se llama cuando el número de elementos de los *chunks* a procesar no es múltiplo del número total de elementos de los vectores. En este caso el último *chunk* tendrá un número de elementos menor al resto y su procesado se tendrá que hacer de manera especial teniendo en cuenta esta circunstancia.

A parte de estos dos ficheros en los que se definen todas las funciones arriba descritas, el programa consta de los siguientes ficheros:

program_kernels.cu: Aquí se definen los kernels del programa.

program_defs.cuh: Este es el archivo de cabecera de nuestro programa. En él se declaran y definen todas las estructuras que más arriba se han comentado, se configuran algunos aspectos de la ejecución del programa (el uso de un fiche de log, por ejemplo) y se declaran todas las funciones de los diferentes archivos en los que se encuentra distribuido el código fuente del programa.

dotproduct_gpu.cpp: Fichero en el que se encuentra el código en C/C++ a partir del cual se crea el archivo mex que llamamos desde MATLAB. En la función mex que se describe en este archivo (*mexFunction*) comprobamos los argumentos de entrada recibidos desde MATLAB, creamos las variables de salida en la que se almacenan los resultados y llamamos a nuestro código en GPU, más concretamente llamamos a la función antes descrita `cudaDotProduct()`.

Capítulo 4

Pruebas y resultados

En este capítulo se realizarán una serie de pruebas sobre el programa desarrollado durante este trabajo, y después se mostrarán los resultados obtenidos haciendo un análisis de los mismos para conseguir obtener ciertas conclusiones acerca del uso de multi-GPU.

4.1. Metodología de las pruebas

Como ya hemos indicado más arriba en este trabajo, además de realizarse el producto escalar de dos vectores, se realiza otro cálculo auxiliar con el único fin de aumentar la carga computacional del algoritmo. La explicación de esto reside en que el cálculo del producto escalar es una tarea demasiado sencilla, que supone poca carga computacional, y entonces la mayor parte del tiempo de cálculo en la GPU se consume en la transferencias de datos y no en el procesamiento propiamente dicho. Para poder comparar los rendimientos de la CPU y la GPU añadimos el cálculo de la siguiente operación a nuestro algoritmo: $C[n] = \sqrt{3{,}14159^n}$, donde n es la posición de cada elemento del vector. Esta operación la realizamos tantas veces como número de elementos tenga nuestros vectores, y de esta manera aumentamos la carga computacional del algoritmo.

Para la realización de las pruebas se han escrito diferentes scripts de MATLAB, desde los que se lanzaban las pruebas con diferentes argumentos, apoyados también por una función propia desarrollada en MATLAB. Esta función recibe como argumentos de entrada los números de elementos de los vectores, el número de elementos por *chunk*, el número de *chunks* que unimos en cada procesamiento, el número de GPUs utilizadas y el número de *streams*; y devuelve como resultados los tiempos que tardan los cálculos en la GPU, en la CPU, y el error relativo entre los resultados de ambos. En esta función se realizan las siguientes tareas:

- Creación dos vectores de valores aleatorios del tamaño que hayamos especificado, que son sobre los que se realizará el cálculo del producto escalar.
- Llamada a la función mex desarrollada, y medir el tiempo que lleva la ejecución de la misma.
- Cálculo del algoritmo en la CPU y temporización de su ejecución. Los cálculos se hacen mediante comandos de MATLAB, luego en realidad estamos comparando con los tiempos que tarda la implementación de MATLAB en la CPU no una implementación nuestra en C/C++

- Cálculo del error relativo entre la implementación en CPU y en GPU.

En los diferentes scripts que se han escrito para lanzar las pruebas se configuraban los distintos argumentos de las simulaciones; se abría un fichero en el que se iban escribiendo los resultados de las simulaciones, que después se utilizarían para hacer las gráficas; y se llamaba a la función arriba descrita con los argumentos pertinentes.

A continuación se muestran los resultados de las distintas pruebas realizadas, y hay que señalar que en lo que sigue de memoria de trabajo siempre que hablemos de CPU nos estamos refiriendo a CPU MATLAB. Esto quiere decir, que los cálculos están realizados en la CPU según la implementación de MATLAB, y no expresamente en un programa en C escrito por nosotros. La implementación del algoritmo en Matlab es el que se muestra en la Figura 4.1:

```
% To increase de duration of the execution
C=zeros(nelements);                              %C is an auxiliar vector
veces=0;
while veces<nelements
    for n = 1 : nelements
        C(n) = sqrt(3.14159^n);
    end
    veces=veces+1;
end

ResultMATLAB = zeros(ceil(nelements/nelementschunk),1);
ProdVecs = vec1.*vec2;
for i = 1:ceil(nelements/nelementschunk)
    if nelements/nelementschunk ~= ceil(nelements/nelementschunk)
       && i == ceil(nelements/nelementschunk)
        ResultMATLAB(i) = sum(ProdVecs(1+(i-1)*nelementschunk : nelements));
        break
    end
    ResultMATLAB(i) = sum(ProdVecs(1+(i-1)*nelementschunk :
                      i*nelementschunk));
end
```

Figura 4.1: Implementación del algoritmo en MATLAB

4.2. Comparación CPU frente a una única GPU sin usar streams

La primera de las pruebas llevadas a cabo consiste en la comparación más simple, es decir comparar el rendimiento en la CPU frente a la GPU. En este primer cálculo en la GPU no se utiliza ninguna de las opciones de *chunks*, GPUs, *streams*, etc., sino que realizamos el procesamiento sin dividir los vectores y utilizando una única GPU sin hacer uso de *streams*. El rango de elementos que abarca esta primera prueba va desde el cálculo

con unos vectores de 10 elementos hasta vectores de 10 elementos, con un incremento gradual entre simulación de 10 elementos.

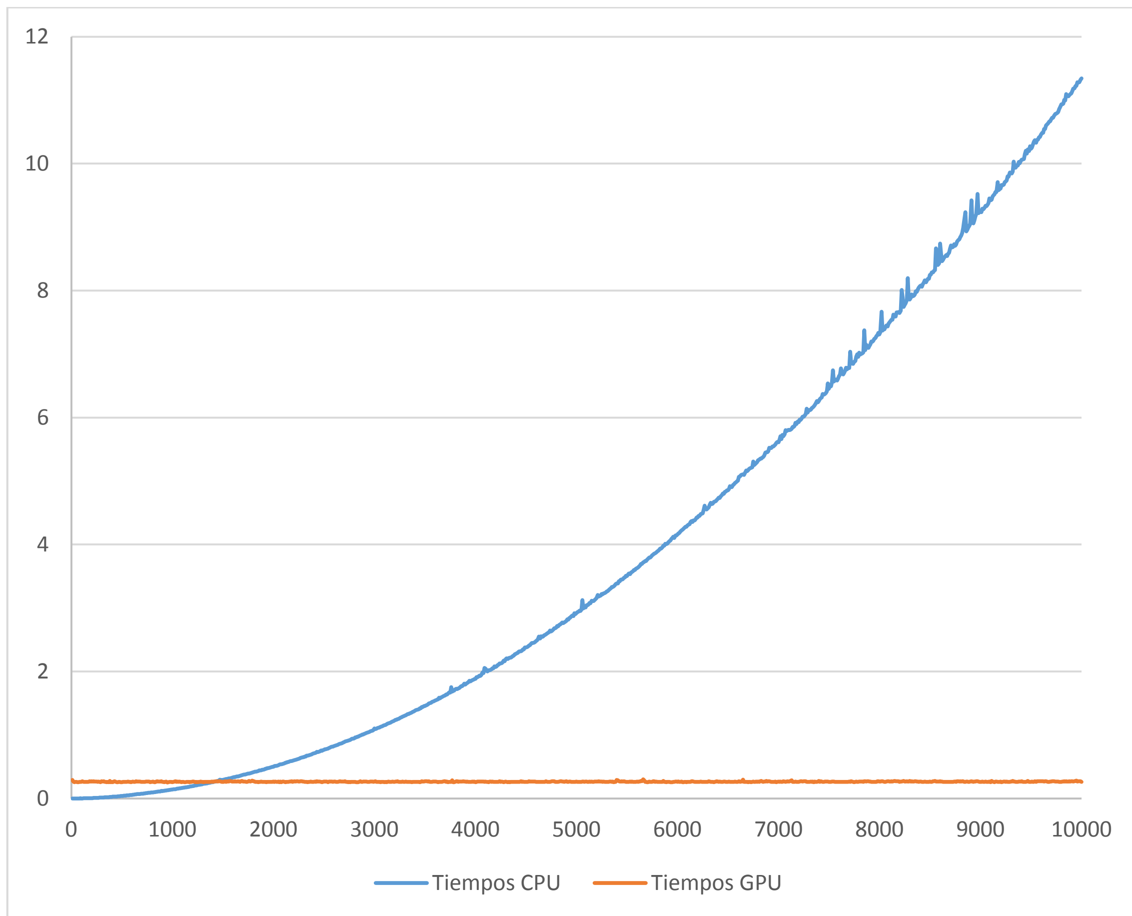


Figura 4.2: Tiempos de ejecución frente a diferentes tamaños de los vectores

En la Figura 4.2 se puede ver cómo, desde el principio, el tiempo de ejecución en la CPU va aumentando gradualmente a medida que aumentamos el número de elementos. A diferencia del tiempo en la GPU que permanece constante para los valores de la prueba. La explicación a este comportamiento puede ser que el número de elementos que tenemos en la prueba no afecta al tiempo que tarda la GPU en el cálculo, es decir que este tiempo que ronda los 0,268 segundos es el tiempo que se tarda en reiniciar las GPUs, hacer las reservas de memoria, etc., y que el tiempo de procesado es despreciable frente a este otro. En sucesivas pruebas veremos cómo vectores de tamaños muy superiores sí afectan a los tiempos de ejecución.

Pese a este comportamiento de la GPU, en el que parece no aumentar el tiempo que tarda en procesar los datos a medida que se aumenta el número de estos, lo cual es en un principio una aspecto ventajoso, se tiene que llamar la atención sobre el hecho de que al

principio de la gráfica se ve cómo la ejecución en la CPU lleva menos tiempo. Si nos fijamos en la Figura 4.3 se puede ver que no merece la pena el uso de la GPU hasta haber superado el tamaño aproximado de 1500 elementos por vector. La explicación a esto es la misma que la expuesta anteriormente, y es que, se podría decir que en nuestro algoritmo el simple hecho de usar la GPU lleva asociado ya una penalización de 0,268 segundos, sea cual sea el número de datos a procesar.

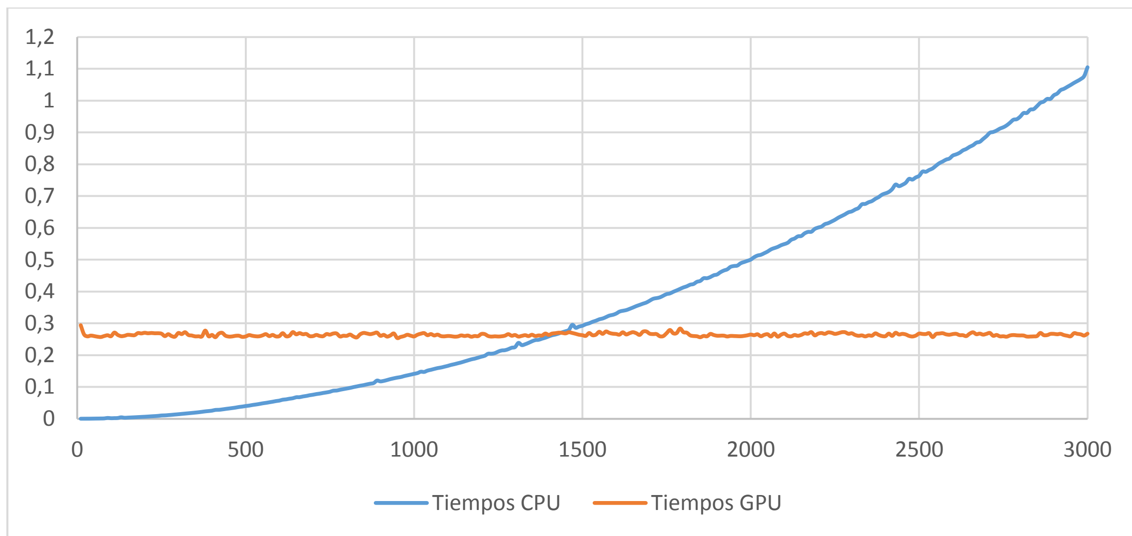


Figura 4.3: Zoom número pequeño de elementos

4.3. Comparación diferentes streams con una GPU

La siguiente prueba realizada consiste en comparar el rendimiento de una única GPU en el cálculo de nuestro algoritmo utilizando diferente número de *streams*. Para ello se han lanzado simulaciones con diferentes tamaños de vectores, empezando con 1024 elementos hasta terminar con 1024000, y variando también el número de *streams* utilizado: 1, 2, 4, 8 y 16 *streams*. No se ha continuado aumentando el número de *streams* ya que 16 es el número de kernels que se pueden ejecutar simultáneamente.

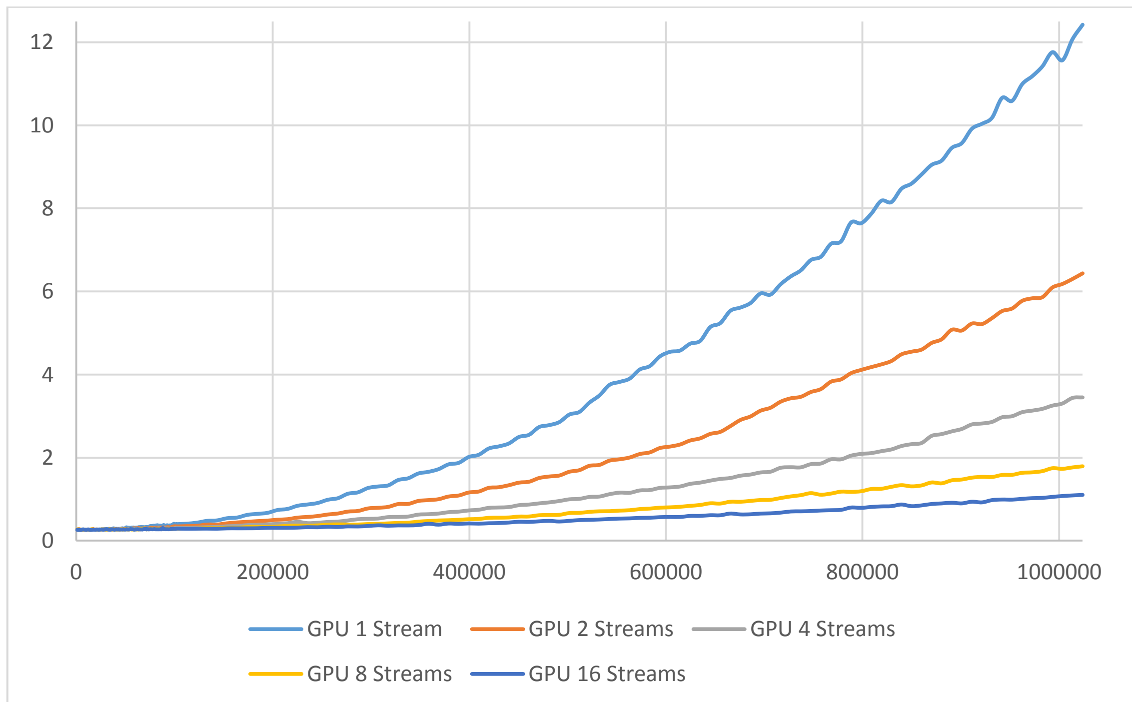


Figura 4.4: Tiempos de ejecución frente a diferentes número de elementos y streams

En la Figura 4.4 se puede ver claramente la mejora que proporciona el aumento del número de *streams* en el procesamiento de datos. Esta mejora sigue una razón tal que doblando el número de *streams* se dobla también el rendimiento, y el tiempo que tardan en realizarse los cálculos disminuye a la mitad. Pero como ya indicábamos más arriba esta tendencia, de doblar el número de *streams* y doblar el rendimiento, no es infinita, pues aunque no hay un límite real en el número de *streams* que se pueden crear, el número máximo de kernels que se pueden ejecutar simultáneamente es de 16. La mejora obtenida utilizando *streams* también dependerá, como ya indicamos en el apartado que hablamos de *streams* (apartado 2.3.7.), de la planificación entre las copias de memoria entre el *host* y el *device*, y las ejecuciones de los kernels.

Cuando el número de elementos no es elevado puede ocurrir que el uso de streams no proporcione ninguna ventaja adicional. En la Figura 4.5 se ve cómo hasta que el número de elementos por vector no ha superado los 40000 o 60000 el utilizar varios streams no reduce el tiempo de ejecución del programa.

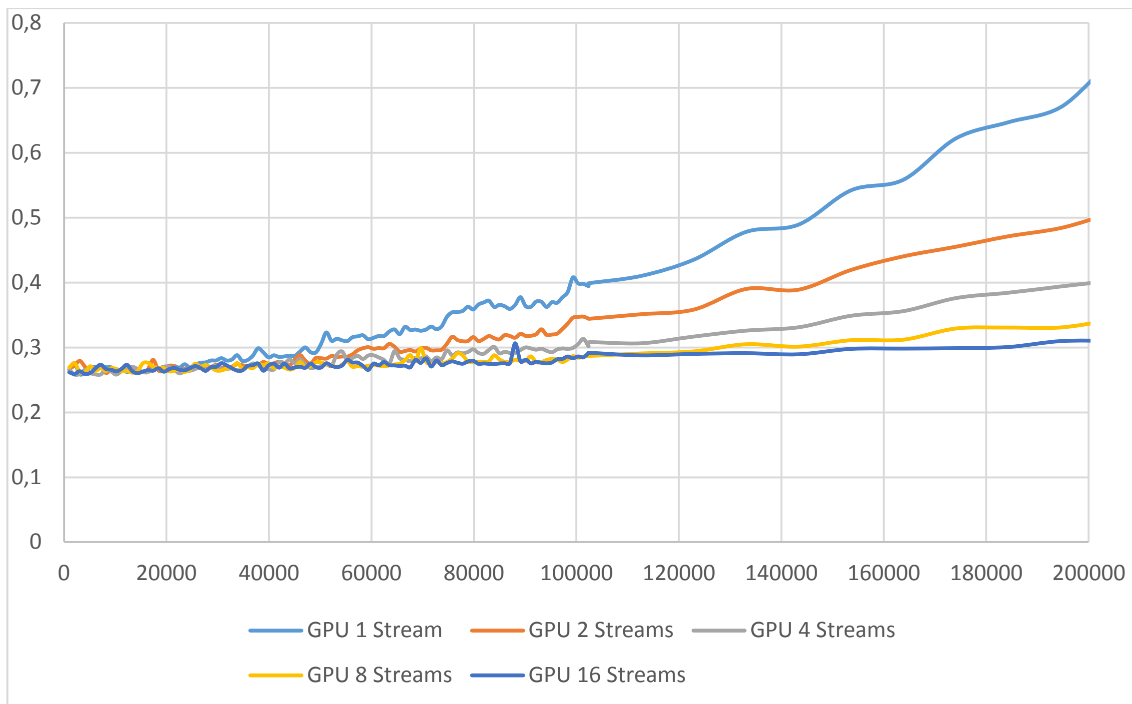


Figura 4.5: Pocos datos y empleo de streams

En la Figura 4.4 solo se aparecían las comparaciones utilizando la GPU, y es natural que surja la pregunta: ¿qué tiempos se obtendrían si se realizaran los cálculos en la CPU? Para hacernos una idea de la increíble diferencia de tiempos entre los cálculos en la CPU y en la GPU se muestran las siguientes figuras (Figura 4.7 y Figura 4.7):

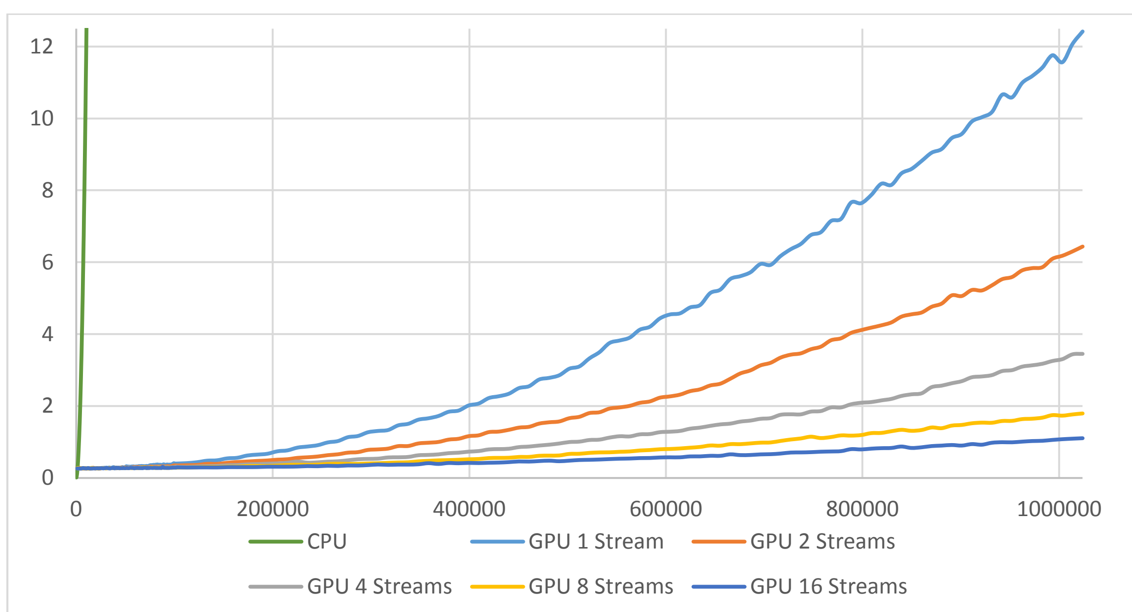


Figura 4.6: CPU vs GPU diferentes streams para vectores grandes

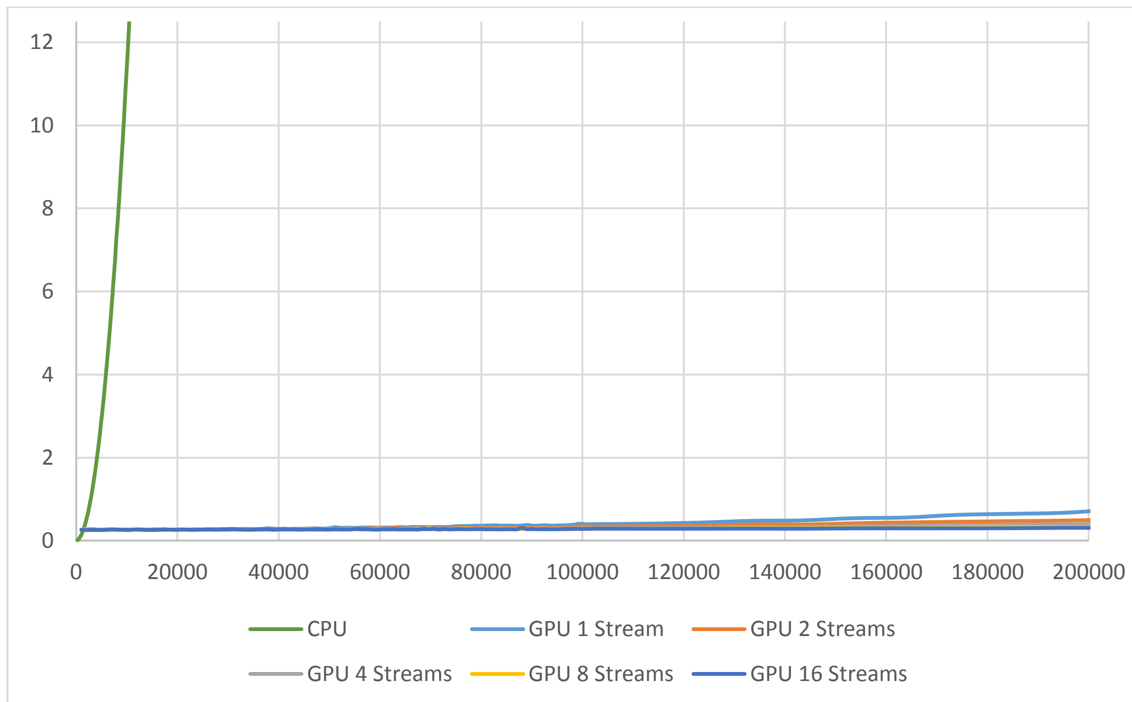


Figura 4.7: Zoom CPU frente GPU diferentes streams

4.4. Comparación una única GPU con multi-GPU

En este apartado se compararán los tiempos de ejecución que obtenemos realizando los cálculos en una única GPU frente a dos GPUs. Debido a que las GPUs instaladas en el servidor son de diferentes modelos, GeForce GTX 770 y GeForce GTX 780, el tiempo que tarda en procesarse un *chunk* del mismo tamaño en cada una de ellas varía. Este hecho no afecta a la medida del tiempo multi-GPU, porque se detiene el temporizador cuando ambas dos han finalizado la ejecución, y entonces la más rápida de ellas está limitada por la otra más lenta. Pero sí que hay que tener en cuenta esta diferencia cuando lo comparemos con la ejecución en una GPU: si para esta prueba se emplease la GPU más rápida la comparación sería desigual, luego, tanto en esta prueba como en las anteriores también, cuando únicamente utilizamos una GPU es la GeForce GTX 770.

La metodología que se ha utilizado para realizar estas pruebas es la misma que en el apartado anterior, barriendo una diferencia de tamaños de vectores que va desde 1024 a 1024000 elementos, y variando también el número de *streams*. Los resultados obtenidos se muestran en la Figura 4.8.

En la Figura 4.8 se puede apreciar que la mejora obtenida por el uso de multi-GPU, con dos GPUs en este caso, frente al uso de una sola es importante. Esta reducción del tiempo de ejecución depende del número de *streams* que empleemos y del tamaño de los

vectores (cantidad de datos a procesar). Por ejemplo, para un único *stream* la mejora obtenida cuando trabajamos con vectores de tamaño 1024000 es de aproximadamente el 68 %, pero sin embargo para ese mismo tamaño de vectores cuando utilizamos 8 *streams* la mejora que se obtiene es del 51 %. De lo que no hay duda es de que a partir de un cierto número de elementos a procesar, la ventaja de utilizar multi-GPU es clara, y que este número de elementos depende a su vez del número de *streams*.

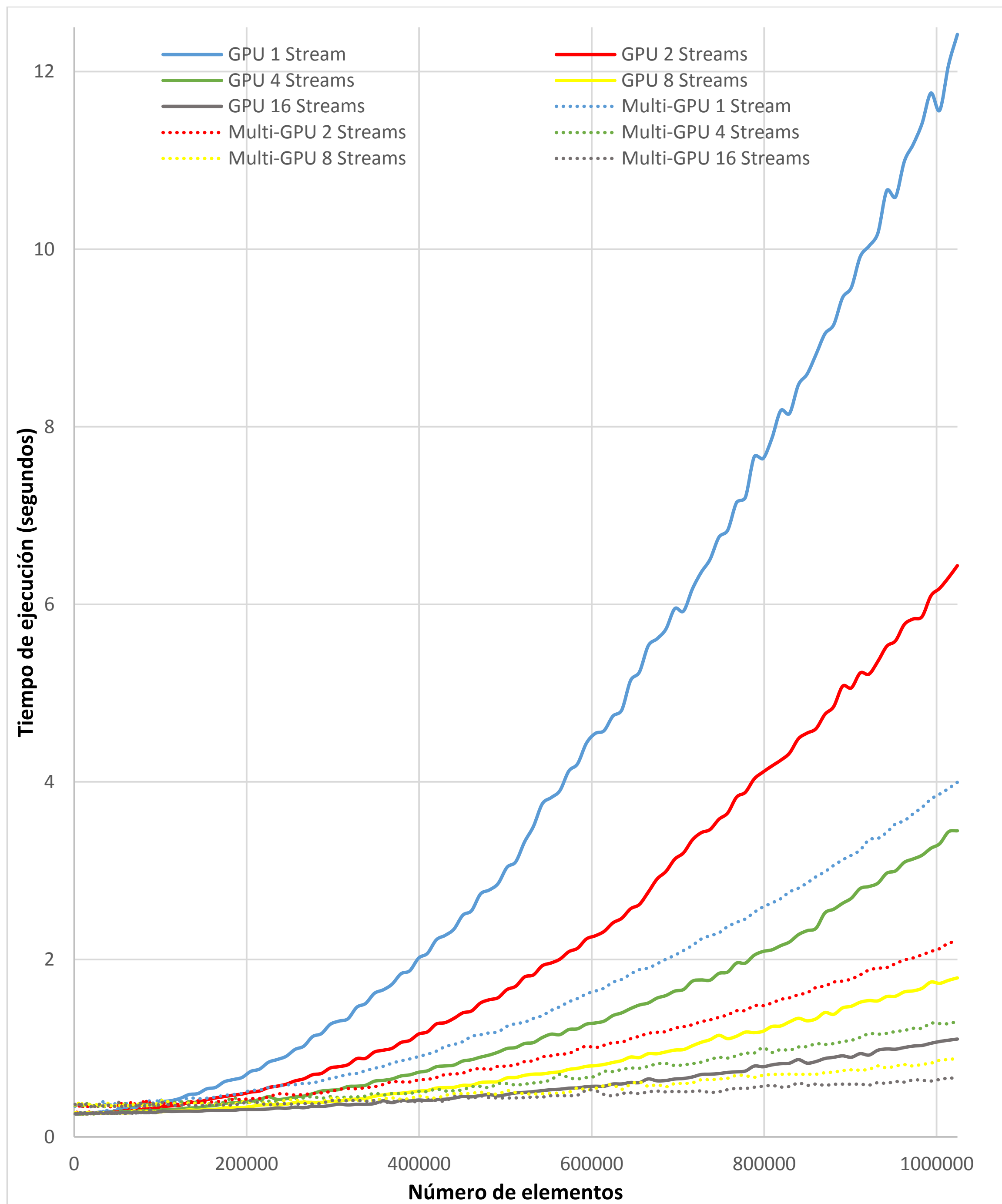


Figura 4.8: Diferentes streams una única GPU y multi-GPU

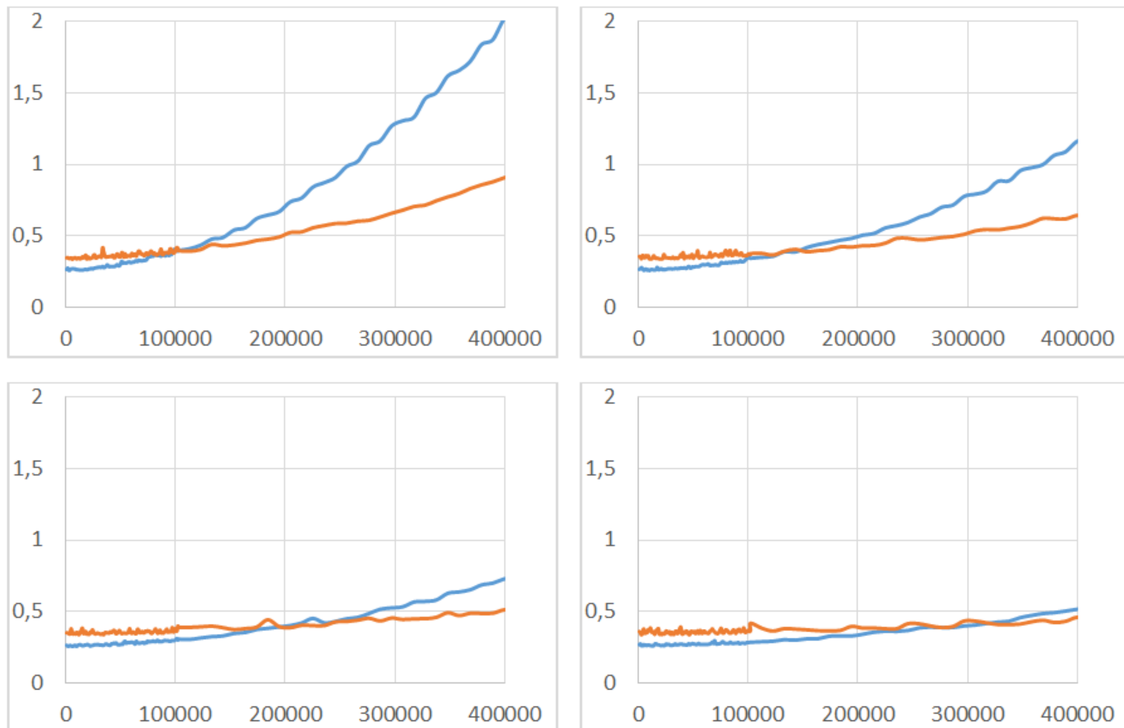


Figura 4.9: Zoom una GPU y multi-GPU utilizando 1, 2, 4 y streams respectivamente

En las gráficas de la Figura 4.9 se tienen los tiempos que tardan en realizarse los cálculos en una GPU frente a varias GPUs para vectores con diferente número de elementos y variando el número de *streams*. Igual que como ocurría cuando se comparó CPU frente GPU, para cantidades de datos no muy grandes lo que en un principio se puede pensar que es una mejora no disminuye el tiempo de ejecución. En este caso se puede ver que hasta que no se supera un cierto tamaño para los vectores, se ejecutan más rápidamente los cálculos utilizando una única GPU que varias. Además, este tamaño mínimo necesario para sacar provecho de multi-GPU aumenta con el número de *streams* que estemos utilizando.

La explicación a este comportamiento es la misma que la del apartado 4.2 (CPU frente a GPU), y es que, si el número de datos no es lo suficientemente grande nuestro algoritmo empleará más tiempo en iniciar las GPUs, realizar reservas de memorias, realizar copias, etc., que lo que es puro procesamiento. En otras palabras, se gasta más tiempo en preparar el procesado de los datos que propiamente procesándolos, y puede ocurrir como vemos en esta Figura 4.9, que otra implementación en principio más lenta resulte luego más rápida porque se ahorra todos esos "preparativos".

4.5. Conclusiones

En este trabajo se ha desarrollado un esqueleto de programa que puede ser reutilizable para futuros trabajos que quieran hacer uso de GPGPU. Para ello se partió de un programa inacabado en el que se podían ver las líneas de trabajo a seguir y esto es lo que se ha hecho. Debido a la forma en que está pensado el programa, separando en distintos bloques lógicos las distintas partes en las que se divide un programa genérico que hace uso de GPGPU, es muy sencilla la reutilización del mismo para otros fines con un número mínimo de cambios.

La operación que se ha implementado en el algoritmo del trabajo es el producto escalar de dos vectores más un cálculo trivial, para aumentar la carga computacional del algoritmo y así poder comparar el rendimiento en la GPU frente a la CPU. Las conclusiones más importantes extraídas de las pruebas realizadas, mostradas en el apartado anterior, son:

- Para que merezca la pena el empleo de GPGPU en el procesado de unos datos, se tiene que tener una cantidad de datos considerable, o la operación de procesado ha de ser de una importante carga computacional.
- Este número de datos a partir del cual se reduce el tiempo de cómputo en comparación con la GPU depende de la operación a realizar. En nuestro caso vimos que era superando el tamaño de 1500 elementos para los vectores de entrada.
- Una vez superado este punto, a medida que aumentamos el número de datos, el ahorro de tiempo obtenido por el uso de GPGPU aumenta exponencialmente.
- El aumento del número de *streams* utilizadas en la GPU, limitado por el número de kernels que es capaz de ejecutar simultáneamente nuestra GPU, supone una disminución en los tiempos de ejecución del programa. Siempre y cuando se supere un mínimo del número de datos, como ocurría con el uso de la GPU frente a la CPU.
- El uso de multi-GPU supone una mejora frente a la utilización de una única GPU, siempre que se supere un mínimo número de datos a procesar.
- La última conclusión, y que en cierta manera engloba a las anteriores, es que una vez que se ha superado el número de datos a partir del cual el uso de la GPU (o del recursos que se haya estado analizando, *streams* o multi-GPU) supone un ahorro frente a la CPU (o la comparativa que corresponda) el ahorro de tiempo consumido en el cálculo del algoritmo continua aumentando a medida que aumenta el número de datos.

Como apéndice a este apartado de conclusiones, indicar que la utilización de ficheros c-mex nos permite programar nuestras propias funciones en MATLAB, y nos permite entonces hacer uso de GPGPU sin necesidad de comprar el Parallel Computing Toolbox.

4.6. Líneas futuras

Por no disponer de la licencia necesaria para la utilización del Parallel Computing Toolbox, quedaría pendiente la comparación de nuestra implementación del algoritmo en GPU y multi-GPU con las implementaciones que hace esta herramienta de cálculos en GPU, en multi-GPU y en multi-CPU (aspecto este último que no ha sido posible tratar en el presente trabajo).

La línea futura principal de este programa es la implementación de diferentes algoritmos utilizando el esquema del programa desarrollado. Haciendo las variaciones necesarias del código, es posible reutilizar gran parte del mismo para ahorrarnos bastante tiempo de trabajo. Sería interesante implementar algún algoritmo con utilidad práctica en cierto trabajo de investigación para acelerar los cálculos. A la par que se reutilizase el esqueleto de programa para nuevos trabajos, el esqueleto de programa se iría adaptando a nuevas necesidades o añadiendo nuevas funcionalidades, y perfeccionándose.

Bibliografía

CASADO GARCÍA, MARIO ENRIQUE. (2010) *Procesamiento Secuencial sobre GPU en CUDA para la Segmentación Bio-Inspirada de Imágenes en Color*. Proyecto fin de carrera. Tutores: Martínez Zarzuela, Mario y Díaz Pernas, Francisco Javier. Universidad de Valladolid.

HENNESSY, JOHN L. y PATTERSON, DAVID A. (1993) *Arquitectura de computadores. Un Enfoque Cuantitativo*. MacGraw-Hill: Morgan Kaufmann Publishers, Inc.

HARRIS, MARK. (2007) *Optimizing Parallel Reduction in CUDA*. NVIDIA Corporation. Disponible desde: https://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf (consultado el 18 de agosto, 2015).

HARRIS, MARK. (2012) *How to Overlap Data Transfers in CUDA C/C++*. NVIDIA Corporation. Blog Parallel ForAll. Disponible desde: http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/ (consultado el 21 de agosto, 2015).

HARRIS, MARK. (2013) *Using Shared Memory in CUDA C/C++*. NVIDIA Corporation. Blog Parallel ForAll. Disponible desde: http://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/ (consultado el 11 de junio, 2015).

KIRK, DAVID B. y HWU, WEN-MEI W. (2010) *Programming Massively Parallel Processors*. A Hands-on Approach. Morgan Kaufmann: Elsevier, Inc.

MARTÍNEZ ZARZUELA, MARIO. (2013) *Presentación del Grupo de Telemática e Imagen para Workshop 3D Technologies*.

MATHWORKS INC. (2015) *Documentación. C/C++ Source Files*. Disponible desde: http://es.mathworks.com/help/matlab/write-cc-mex-files.html (consultado el 25 de agosto, 2015)

MICIKEVICIUS, PAULIUS. (2011) *Multi-GPU Programming*. NVIDIA Coporation. Disponible desde: http://www.nvidia.com/docs/io/116711/sc11-multi-gpu.pdf (consultado el 21 de agosto, 2015).

NVIDIA CORPORATION. (2009a) *CUDA C Programming Guide v 2.3.1*. Disponible desde:
http://www.cs.colostate.edu/~cs675/cudaDocs/NVIDIA_CUDA_Programming_Guide_2.3.pdf (consultado el 18 de agosto, 2015).

NVIDIA CORPORATION. (2015a) *CUDA C Programming Guide v 7.0*. Disponible desde: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (consultado el 18 de agosto, 2015).

NVIDIA CORPORATION. (2015b) *CUDA Runtime API v 7.0*. Disponible desde: http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf (consultado el 18 de agosto, 2015).

NVIDIA CORPORATION. (2015c) *NVIDIA GeForce GTX 770 GPU datasheet*. Diponible desde: http://www.nvidia.es/object/geforce-gtx-770-es.html#pdpContent=2 (consultado el 10 de agosto, 2015).

NVIDIA CORPORATION. (2015d) *NVIDIA GeForce GTX 780 GPU datasheet*. Disponible desde: http://www.nvidia.es/object/geforce-gtx-780-es.html#pdpContent=2 (consultado el 10 de agosto, 2015).

PHARR, MATT. (2005) *GPU Gems 2. Programming Techniques for High-Performance Graphics and General Purpose Computation*. NVIDIA Corporation: Pearson Education, Inc. Disponible desde:
https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_inside_front_cover.html (consultado el 13 de agosto, 2015).

QINGANG XIONG, BO LI, JI XU, XIAOWEI WANG, LIMIN WANG, WEI GE. (2012) *Efficient 3D DNS of gas–solid flows on Fermi GPGPU*. Computers & Fluids, Volume 70, Issue null, Pages 86-94.

SANDERS, JASON y KANDROT, EDWARD. (2011) *CUDA By Example. An Introduction to General-Porpose GPU Programming*. Addison-Wesley: Pearson Education, Inc.

TEJERO DE PABLOS, A. (2009) *Implementación sobre GPU en CUDA de redes neuronales artificiales basadas en la teoría de resonancia adaptativa*. Proyecto fin de carrera. Tutores: Martínez Zarzuela, M., Diaz Pernas, F. J. Universidad de Valladolid.

TRAPNELL, COLE y C. SCHARTZ, MICHAEL. (2009) *Optimizing data intensive GPGPU computations for DNA sequence alignment*. Parallel Computing, Volume 35, Issue 8, Pages 429-440.

WILT, NICHOLAS. (2013) *The CUDA Handbook. A Comprehensive Guide to GPU Programming*. Addison-Wesley: Pearson Education, Inc.

W. SUH, JUNG y KIM, YOUNGMIN. (2014) *Accelerating MATLAB with GPU Computing.* A Pimer with Examples. Morgan Kaufmann: Elsevier, Inc.