REPUBLICA BOLIVARIANA DE VENEZUELA

UNIVERSIDAD VALLE DEL MOMBOY

FACULTAD DE INGENIERÍA UVM

CARVAJAL ESTADO TRUJILLO

ACELERACION GRÁFICA POR GPU PARA OPTIMIZAR EL RENDIMIENTO EN DIFERENTES ORDENADORES

Carvajal, Junio 2018

REPUBLICA BOLIVARIANA DE VENEZUELA

UNIVERSIDAD VALLE DEL MOMBOY

FACULTAD DE INGENIERÍA UVM

CARVAJAL ESTADO TRUJILLO

ACELERACION GRÁFICA POR GPU PARA OPTIMIZAR EL RENDIMIENTO EN DIFERENTES ORDENADORES

Trabajo Especial de Grado para optar al Grado de
Ingeniería en Computación

Autor: Yonmar Steven Villarreal Páez

Tutor: Dr. Iván Pérez

Carvajal, Junio 2018

DEDICATORIA

A ***mis padres, Yonny Villarreal y Mary Páez***: Por permitirme ser quien soy, ustedes son un gran ejemplo de vida y esfuerzo, y acá están sus recompensas, este logro es de ustedes!

A mi hermana ***Yonmairy*** que parte de ti me ha enseñado ser un emprendedor, siempre has estado a mi lado en las buenas y malas, y me has presentado antes los demás por el conocimiento de mi profesión. Este triunfo también es tuyo y espero que te sirva como ejemplo para un mayor impulso en tu carrera y vida, Te quiero.

A mi Tío ***Juan Carlos***: Gracias por haberme dado la gran oportunidad de ser un empleado a los largo de los años de mis estudios, los cuales fueron sin duda el combustible que mantenía este motor en impulso para lograr esta meta y eres como un segundo padre que siempre me dio la mano para salir adelante.

A mis familiares, ***Tío Richard, Tía Zulay, Abuela Mery***, Les dedico este triunfo en alto y les doy las gracias por creen en mí, en lo que me propongo, por hacerme saber lo mucho que valgo y las cosas tan increíbles que soy capaz de hacer. A mi abuelo ***Juan*** que desde cielo sé que me proteges y estas orgulloso por este triunfo!

A ***Rodolfo Barrios***, quien siempre ha sido ese único amigo hermano que me dio la vida incondicionalmente y lamentablemente por motivos de situaciones del país nos tocó ir por diferentes caminos, Gracias por tu apoyo a distancia, por creer en mí, espero compartir este triunfo contigo presentemente en un futuro, ya que parte de este logro te pertenece y te lo dedico por haberme decido a escoger y empezar esta carrera contigo al inicio.

Y finalmente esta dedicatoria es en especial ***a mí***, por haber creído desde un principio en mí mismo, en que todo lo iba a lograr sin importar los obstáculos, tiempo, o sacrificios, como siempre he dicho y diré, cada sacrificio trae su recompensa, y este es el tuyo ***Yonmar!***

AGRADECIMIENTO

A ***Dios***: principalmente por haberme dado toda la fuerza, sabiduría para lograr esta meta y permitirme compartirla con todos mis seres queridos aun en tiempos difíciles pero teniendo una fe en que todo se puede lograr.

A la ***Universidad Valle del Momboy*** por permitirme prepararme bajo sus enseñanzas.

A la ***Clínica María Edelmira Araujo***, por haberme permitido realizas mis practicas profesiones en dicha institución y aun así terminándome de abrir las puertas para seguir ejerciendo y reconociendo mi esfuerzo y profesión.

A mis Padres, ***Yonny Villarreal*** y ***Mary Páez*** por ser ese motor de arranque, que desde el primer día en que inicié clases, me facilitaron las puertas para iniciar mis estudios así como para finalizar mis estudios, ustedes me enseñaron en ir adelante y a luchar por lo que quiero, no me alcanzará la vida para agradecerles todo el apoyo que me han dado ¡los amo, este triunfo es de ustedes!

A mis colegas de estudios, ***Luis Paredes, Alfredo Martínez***, ***Andrés Terán, Raúl Materán, José González, Javier Graterol, Jorge Avellaneda,*** que a pesar de que soy muy selectivo con las amistades, ustedes aportaron apoyo y esfuerzo a lo largo de la carrera, además de compartir y ayudarnos mutuamente en sus respectivos momentos.

A mis amistades ***Edgardo Altuve, Johanna Camacho, Alejandro Mendoza, Roberth Suarez, Yohan V, Estefani Paredes, Jeison Viloria,*** gracias por el valor que me han dado en estos últimos tiempos, es bueno saber que ustedes reconocen mi esfuerzo y lo que soy hoy en día. A la cachetona ***Leidy Tatiana,*** Gracias por creer en mí, por tu apoyo, por toparte conmigo en una etapa de vida en donde se necesita apoyo mutuo.

A mi tutora laboral de Prácticas Profesionales ***Carmen Muchacho,*** por facilitarme las puertas en su área de trabajo y facilitarme a tiempo la entrega de este proyecto.
Mi agradecimiento especial, a mi Tutor, ***Iván Pérez,*** por sus enseñanzas como profesor, por aceptarme este proyecto y brindarme todo su apoyo y su conocimiento.

ÍNDICE

DEDICATORIA iv
AGRADECIMIENTO v
ÍNDICE vi
RESUMEN ix
CAPÍTULO I 1
1.1. MARCO INTRODUCTORIO 1
1.2. PLANTEAMIENTO DEL PROBLEMA: 2
1.3. FORMULACIÓN DEL PROBLEMA 4
1.4. OBJETIVO DE LA INVESTIGACIÓN 4
1.4.1. OBJETIVO GENERAL 4
1.4.2. OBJETIVOS ESPECÍFICOS 5
1.5. JUSTIFICACION DE LA INVESTIGACIÓN 5
1.6. DELIMITACION DE LA INVESTIGACIÓN 5
CAPÍTULO II 7
2.1. MARCO TEORICO 7
2.2. ANTECEDENTES DE LA INVESTIGACIÓN 7
2.3. BASES TEÓRICAS 9
CAPÍTULO III 11
3.1. MARCO METODOLÓGICO 11
3.2. DISEÑO DE INVESTIGACIÓN 11
3.3. MÉTRICAS DE DESEMPEÑO 12
3.3.1. TIEMPOS DE EJECUCIÓN 13
3.3.2. ACELERACIÓN 13
3.4. TIPOS DE ARQUITECTURAS 14
3.4.1. ARQUITECTURAS PARALELAS 14
3.4.1.1. MECANISMOS DE CONTROL 14
3.4.2. ARQUITECTURAS DEL SISTEMA DE LA GPU 17
3.4.2.1. ARQUITECTURA DEL SISTEMA HETEROGÉNEO CPU-GPU 17
3.4.2.2. ARQUITECTURA DE MEMORIA GPU UNIFICADA (UMA): 19

3.4.2.3. SISTEMA MULTI-GPU SLI 20
3.4.3. ARQUITECTURA DEL PIPELINE 20
3.4.4. ARQUITECTURAS ORIENTADAS AL PROCESAMIENTO GRÁFICO 23
3.4.4.1. ARQUITECTURA NVIDIA 23
3.4.4.2. ARQUITECTURA AMD (ATI) 24
3.4.4.3. ARQUITECTURA CUDA: 25
3.4.4.4. ARQUITECTURA DE OPENCL 27
3.5. DIFERENCIAS, SIMILITUDES E INTERCONEXIÓN ENTRE LAS ARQUITECTURAS CPU Y GPU. 28
3.5.1. CPU vs GPU 31
3.5.2. DEFINICIÓN DE HOST Y DEVICE 31
3.5.3. Hilos y RAM 31
CAPÍTULO IV 32
4.1. PROGRAMACIÓN DE LAS GPU 32
4.1.1. CÁLCULO PARALELO EN GPU: 32
4.2. MODELO DE PROGRAMACIÓN CUDA 33
4.3. MODELO DE MEMORIA 37
4.3.1. MEMORIA GLOBAL 38
4.3.2. MEMORIA LOCAL 39
4.3.3. REGISTROS 40
4.3.4. MEMORIA COMPARTIDA 40
4.3.5. MEMORIA CONSTANTE 41
4.3.6. MEMORIA DE TEXTURAS 41
4.4. ETAPAS PROGRAMABLES DEL PIPELINE 42
4.4.1. INTERFACES DE PROGRAMACIÓN DEL PIPELINE GRÁFICO 44
4.4.2. OpenGL 44
4.4.3. Direct3D 46
4.5. ESTRUCTURA DE UN PROGRAMA EN CUDA 47
4.5.1. TRANSFERENCIA DE DATOS CPU-GPU 51

4.6. ESTRATEGIAS Y OPTIMIZACIONES 54
4.6.1. OPTIMIZACIONES DE PÍXELES 54
4.6.2. OPTIMIZAR USO DE MEMORIA 55
4.6.3. FLUJO DE INSTRUCCIONES 56
CAPÍTULO V 60
5.1. COMPARACIÓN DE RENDIMIENTO ENTRE LA CPU Y GPU 60
5.2. COMPARACIÓN ENTRE EL PIPELINE SECUENCIAL (IZQUIERDA) Y EL CÍCLICO DE LA ARQUITECTURA UNIFICADA (DERECHA) 62
5.3. COMPARACION DE MODELOS DE NVIDIA Y AMD 63
5.4. OPERACIONES DE PUNTO FLOTANTE POR SEGUNDO GPU vs CPU 64
5.4.1. PROCESAMIENTO DE IMAGENES 66
5.5. APLICACIONES DE CUDA 66
5.5.1. Aplicaciones de CUDA (Energía): 66
5.5.2. Aplicaciones de CUDA (Finanzas) 67
5.5.3. Aplicaciones de CUDA (Investigación) 68
5.5.4. Aplicaciones de CUDA (Medicina) 69
5.5.5. Aplicaciones de CUDA (Vídeo y Fotografía) 69
5.6. PRESTACIONES CPU-GPU 70
5.6.1. PRESTACIONES DE CPU/GPU EN CÁLCULO Y ACCESO A MEMORIA 73
5.7. Evolución GPU: GB/s. 73
CONCLUSIONES 74
RECOMENDACIONES 76
BIBLIOGRÁFICA 77

Autor:
Br. Yonmar Villarreal
C.I. 23.594.995
Tutor: Dr. Iván Pérez
Año: 2018

REPUBLICA BOLIVARIANA DE VENEZUELA
UNIVERSIDAD VALLE DEL MOMBOY
FACULTAD DE INGENIERÍA UVM
CARVAJAL ESTADO TRUJILLO

ACELERACION GRÁFICA POR GPU PARA OPTIMIZAR EL RENDIMIENTO EN DIFERENTES ORDENADORES

Autor: Yonmar Steven Villarreal Páez
Tutor: Dr. Iván Pérez
Fecha: Junio, 2018

RESUMEN

La investigación presentada tuvo como objetivo estudiar cómo actúa la aceleración grafica por GPU para obtener mejores resultados de rendimientos en los ordenadores que requieran trabajar con fluidez en cuanto al área de diseño gráfico o bien sea con el procesamiento de cargas gráficas en el computador. Para su desarrollo se aplicó la metodología de investigación no experimental descriptiva, donde se obtuvo diferentes variables importantes que conforman el estudio y evolución de las diferentes arquitecturas orientadas a la computación gráfica, desde dispositivos especializados en su tarea para la generación de gráficos hasta dispositivos computacionales masivamente paralelos aptos para la computación de otras prestaciones de propósito general. Los resultados fueron obtenidos y presentados mediantes graficas comparativas entre el rendimiento de una GPU vs CPU, arquitecturas orientadas a la computación graficas por medio del *pipeline* gráfico. Finalmente hemos identificado el tipo de aplicaciones que se pueden sacar provecho a la implementación de CUDA en las GPU. Las recomendaciones finales van dirigidas al programador para optar por un mejor rendimiento.

Palabras Clave: Arquitecturas, Gráficos, Procesador, Programación, GPU, CPU, CUDA, Pipeline, OpenGL, Direct3D, Nvdia, AMD, Intel.

CAPÍTULO I

1.1. MARCO INTRODUCTORIO

Actualmente existe un conjunto de tecnologías de procesamiento masivo de datos que ayudan para la automatización de la resolución de problemas de cualquier índole. Estos problemas, dependiendo del número de operaciones que se realizan, pueden tardar un tiempo prolongado para su resolución.

En este módulo didáctico, vamos a estudiar las arquitecturas basadas en computación gráfica (GPU), que es una de las tendencias en computación paralela con más crecimiento en los últimos años y que goza de gran popularidad, entre otras cuestiones, debido a la buena relación entre las prestaciones que ofrecen y su coste.

Primero repasaremos los fundamentos de la computación gráfica para entender los orígenes y las características básicas de las arquitecturas basadas en computación gráfica. Estudiaremos el pipeline gráfico y su evolución desde arquitecturas con elementos especializados hasta los pipelines programables y la arquitectura unificada, que permite la programación de aplicaciones de propósito general con GPU. Veremos las diferencias principales entre CPU y GPU y algunas de las arquitecturas GPU más representativas junto a sus aplicaciones que fueron introduciéndose gracias a la implementación CUDA, poniendo énfasis en cómo pueden ser programados ciertos elementos.

Una vez presentadas las arquitecturas orientadas a computación gráfica, nos centraremos en las arquitecturas unificadas modernas que están focalizadas en la programación de aplicaciones de propósito general. Analizaremos las características de las principales arquitecturas relacionadas, como son las de Nvidia, AMD e Intel. Finalmente, estudiaremos CUDA y OpenCL, que son los principales modelos de programación para aplicaciones de propósito general sobre GPU.

1.2. PLANTEAMIENTO DEL PROBLEMA:

La evolución de los sistemas de computación con multiprocesadores han seguido dos líneas de desarrollo: las arquitecturas *multicore* (*multi-núcleos*) y las arquitecturas *many-cores* (*muchos-núcleos*).

En un computador genérico de arquitectura (*multicore)* no es posible reemplazar la CPU por una GPU, puesto que La Unidad Central de Procesamiento (*Central Processor Unit*, CPU), está diseñada únicamente para realizar operaciones matemáticas, lógicas con una frecuencia de reloj medida en Ghz (*Gigahercios*) para que las aplicaciones logren ser ejecutadas con su propósito general. Cuando se ejecutan aplicaciones con cargas gráficas, la CPU requiere más potencia y se ve obligada distribuir sus recursos entre las tareas ejecutadas, esto origina el congestionamiento de su función principal y disminuye su rendimiento.

En el caso de las arquitecturas many-core, los desarrollos se centran en optimizar el desempeño de las aplicaciones, dentro de este tipo de arquitectura se encuentran las Unidades de Procesamiento Gráfico (Graphics Processing Unit, GPU)

Al incorporarse un procesador GPU, el cual se especializa únicamente en gráficos, se libera la carga de trabajo gráfica de la CPU, lo que permite mejor rendimiento para implementar instrucciones más complejas.

En la actualidad *multicore* (*multi-núcleos*). Son hechas para proveer mejor desempeño a las soluciones secuenciales, es por ello que las optimizaciones se basan por ejemplo en, proveer lógica de control compleja para la ejecución paralela de códigos secuenciales o incluir memorias caché más rápidas y más grandes para disminuir la latencia de las instrucciones y así ejecutar todo tipo de aplicaciones (de ahí su propósito general)

Los *multicore* se iniciaron con sistemas de 2 núcleos y con cada generación se duplica este número, actualmente el procesador Intel *Core* i7 cuesta con versiones de 2 a 6 núcleos, en ellas se encuentra el poder de procesamiento. Hoy en día ya no

existen computadoras con un único procesador, es más ya está por desaparecer las de 2 *cores*.

Las Unidades de Procesamiento Gráfico, GPU, generan gráficos 2D y 3D, imágenes y video que posibilitan el desarrollo de sistemas operativos basados en ventanas, interfaces gráficas de usuario, videojuegos, aplicaciones de formación y representación de imágenes visuales y video. Las GPU Puede utilizarse como un procesador gráfico programable o como un sistema paralelo escalable que proporciona una interacción visual en el tiempo real con objetos compuestos de gráficos, imágenes y video.

En los últimos años, su evolución implico un cambio, dejó de ser un procesador gráfico potente para convertirse en un co-procesador apto para el desarrollo de aplicaciones paralelas de propósito general con demanda de anchos de banda de procesamiento y de memoria sustancialmente superiores a los ofrecidos por la CPU. Por ello actualmente la GPU se ha posicionado como una alternativa atractiva a los sistemas tradicionales de computación paralela ya que su objetivo es liberar a la CPU del proceso de renderizado propio de las aplicaciones gráficas.

Los PCs y las videoconsolas combinan la GPU con una CPU para formar un sistema heterogéneo, en las PCs la comunicación entre la CPU y GPU se lleva a cabo por medio de un puerto llamado PCI Express o PCIe (Peripheral Component Interconnect Express) el cual es el estándar para ejecutar esta comunicación

La CPU son diseñada para ejecutar una sola instrucción en un dato tan pronto como sea posible, es por ello que la CPU es incapaz de ofrecer un rendimiento a la altura del resto del sistema, por lo tanto frena y afecta el rendimiento de otros componentes, impidiendo que éstos puedan desarrollar todo su potencial.

El procesamiento de la CPU es en serie, se compone de unos pocos núcleos muy complejos que pueden ejecutar unos pocos programas al mismo tiempo y esto afecta al rendimiento óptimo, mientras que la GPU es un procesador de propósito

específico que está optimizada para trabajar con grandes cantidades de datos y realizar las mismas operaciones, una y otra vez por lo que la GPU puede permitirse el lujo de implementar instrucciones complejas.

1.3. FORMULACIÓN DEL PROBLEMA

Uno de los principales inconvenientes a la hora de trabajar con GPU es la dificultad para el programador a la hora de transformar programas diseñados para CPU tradicionales en programas que puedan ser ejecutados de manera eficiente en una GPU. Por este motivo, se han desarrollado modelos de programación, ya sean de propiedad (CUDA) o abiertos (OpenCL), que proporcionan al programador un nivel de abstracción más cercano a la programación para CPU, que le simplifican considerablemente su tarea.

Por otro lado la solución que la industria adoptó fue el desarrollo de procesadores con múltiples núcleos que se centran en el rendimiento de ejecución de aplicaciones paralelas en contra de programas secuenciales. Así pues, en los últimos años se ha producido un cambio muy significativo en el sector de la computación paralela. En la actualidad, casi todos los ordenadores de consumo incorporan procesadores multinucleo. Desde la incorporación de los procesadores multinucleo en dispositivos cotidianos, desde procesadores duales para dispositivos móviles hasta procesadores con más de una docena de núcleos para servidores y estaciones de trabajo, la computación paralela ha dejado de ser exclusiva de supercomputadores y sistemas de altas prestaciones. Así, estos dispositivos proporcionan funcionalidades más sofisticadas que sus predecesores mediante computación paralela.

1.4. OBJETIVO DE LA INVESTIGACIÓN

1.4.1. OBJETIVO GENERAL

Evaluar la influencia del rendimiento por causa de la aceleración de GPU teniendo en cuenta las prestaciones de los distintos componentes de hardware en el computador.

1.4.2. OBJETIVOS ESPECÍFICOS

1. Identificar los funcionamientos de las arquitecturas gráficas y características básicas del *pipeline* gráfico.
2. Describir las diferencias y similitudes entre las arquitecturas CPU y GPU.
3. Interpretar los conceptos fundamentales e implementación de las Arquitecturas Orientadas al Procesamiento Gráfico.
4. Presentar los resultados de rendimiento, beneficios y optimización.

1.5. JUSTIFICACION DE LA INVESTIGACIÓN

Anteriormente en las décadas de los 90 el procesamiento de gráficos en un PC se realizaba por medio del controlador VGA (*Video Graphics Array*), era simplemente un controlador de memoria y un generador de visualización conectado a una DRAM, poco a poco la tecnología fue avanzando lo suficiente para añadir más funciones en el controlador VGA y empiezan a incorporarse algunas funciones de aceleración como 3D, al mismo tiempo empresas como Nvidia, Ati Technologies y 3dfx Interactive, empezaron a comercializar aceleradores gráficos capaces de implementar cada vez más etapas del *pipeline* gráfico directamente en el procesador gráfico. Desde ese entonces Nvidia representó el cambio más significativo en la Tecnología GPU.

Es por ello, que esta investigación tiene como objetivo principal estudiar cómo influye el rendimiento óptimo en un computador por causa de la aceleración de GPU, de tal manera en que esto beneficiará a todas aquellas televisoras o empresas de diseños gráficos que trabajan con computadores que requieren cargas de alto gráfico para la realización de un proyecto.

1.6. DELIMITACION DE LA INVESTIGACIÓN

La presente investigación se delimitara a la implementación de la Arquitectura de las GPU, conociendo los distintos modelos de programación de CUDA y OpenGL,

así como también conocer las Arquitecturas orientadas de propósito general sobres las GPU como lo son las Arquitecturas NVDIA y AMD (ATI).

Con la finalidad de enfocar este conocimiento y aplicarlo exclusivamente en computadores que se requerirán un alto procesamiento de gráficos y de este modo observar el rendimiento y optimización, como lo pueden ser las empresas de televisora de nuestra ciudad (Tv Andes, Valera Noticias), o empresas especializados en el área de Fotografía y Edición de Videos (Impresas C.A, Publicorp C.A).

CAPÍTULO II

2.1. MARCO TEORICO

A partir del 2005 se comenzó a utilizar a la gran potencia de cálculo y el alto número de procesadores de las GPU como arquitectura masivamente paralela para resolver tareas no vinculadas con actividades gráficas, es decir utilizarlas en programación de aplicaciones de propósito general (GPGPU). En este ámbito surgieron varias técnicas, leguajes y herramientas para la programación de GPU como co-procesador genérico a la CPU. Una de las herramientas más difundidas es CUDA (Compute Unified Device Architecture), desarrollada por Nvidia para sus GPU desde la generación G80.

En la actualidad las plataformas multicore lideran la industria de los computadores, obligando a los desarrolladores software a adaptarse a nuevos paradigmas de programación para poder explotar su capacidad de cómputo. A día de hoy uno de los principales exponentes de las plataformas multicore son las unidades de procesamiento gráfico (GPUs).

2.2. ANTECEDENTES DE LA INVESTIGACIÓN

La arquitectura Von Neumann, también conocida como modelo de Von Neumann o arquitectura Princeton, es una arquitectura de computadoras basada en la descrita en 1945 por el matemático y físico John Von Neumann y otros, en el primer borrador de un informe sobre el EDVAC.

Este describe una arquitectura de diseño para un computador digital electrónico con partes que constan de una unidad de procesamiento que contiene una unidad aritmético lógica y registros del procesador, una unidad de control que contiene un registro de instrucciones y un contador de programa, una memoria para almacenar tanto datos como instrucciones, almacenamiento masivo externo, y mecanismos de entrada y salida.

En los primeros ordenadores el procesador central (CPU), era el encargado de gestionar y procesar todo tipo de información. Desde los datos que el usuario quería operar hasta por supuesto el sistema operativo, y con él su interfaz.

Si bien aquellos primeros sistemas utilizaban interfaces basadas en texto, con la llegada de las primeras interfaces gráficas el nivel de exigencia creció no sólo en el propio sistema operativo, sino también en muchas de las aplicaciones que empezaban a surgir por la época. Programas CAD o videojuegos, por ejemplo, requerían muchos más recursos para funcionar correctamente.

Llegados a este punto, los diseñadores de sistemas se basaron en un componente que ya existía para evolucionarlo y hacerlo crecer. Los coprocesadores matemáticos o FPU, Floating Point Únit, eran utilizados en muchos sistemas para acelerar el procesamiento de datos. Pueden entenderse como un segundo procesador, si bien algunas de las diferencias respecto de las CPU son muy claras: no pueden tener acceso a los datos directamente (debe ser la CPU la que gestione este apartado) o ejecutan un juego de instrucciones mucho más sencillo pensado para tratar datos en coma flotante.

Las exigencias continuaron creciendo, y los sistemas de la época disponían de CPU y una FPU optativa que terminó convirtiéndose en fundamental: los coprocesadores matemáticos evolucionaron hacia las GPU, al ser el componente más eficiente a la hora de procesar y determinar el aspecto gráfico de todo tipo de software.

Los coprocesadores matemáticos siguieron evolucionando y mejorando, y empezaron a montarse en tarjetas individuales. Mediante este formato podían disponer de un mayor espacio para crear chips más grandes, con más transistores y circuitería y mejores conexiones energéticas, que eran capaces de ofrecer una mayor capacidad de proceso.

No fue hasta 1.999 cuando NVidia acuñó el término GPU, Graphics Processing Unit, para sustituir a las anteriores *tarjetas de vídeo*. Tras unas exitosas *RIVA TNT2* presentaron la NVidia GeForce 256, y para promocionarla pusieron gran énfasis en las posibilidades gráficas que aportaba a nuestro equipo. Los videojuegos, ganando cada vez más adeptos, fueron una de las claves para que los diseñadores de las GPU fueran incrementando su rendimiento año tras año.

En dicho proyecto se ilustran los resultados obtenidos sobre el impacto de distintas transformaciones de código de alto nivel en el rendimiento de distintos algoritmos en la GPU.

Sobre este particular, se consideró de interés conocer las estrategias de optimizas el uso de memoria, así como los distintos acceso de memorias como lo son, la memoria compartida, memoria global, memoria local, memoria constante y memoria de texturas.

2.3. BASES TEÓRICAS

A continuación se presentan las bases teóricas que sustentan al estudio sobre el rendimiento de computador por la aceleración grafica GPU.

El estudio se relaciona con varias variables que le dan forma y se vincula con el proyecto planteado. Sobre este particular, Francesc Guim (2011), menciona que lo principal es entender los orígenes y características básicas de las arquitecturas basadas en la computación gráfica (*pipeline*) y así mismo las diferencias entre CPU y GPU poniendo énfasis en como pueden ser programadas, en ese sentido, las siguientes variables se consideran adecuadas para fundamentar tal instrumentación y sustentar la línea de investigación de David A. Patterson y John L. Hennessy (2011), donde da a conocer el modelo de programación de aplicaciones graficas en la GPU las cuales corresponde a DirectX, OpenGL, y CUDA. Dichas variables las daremos a conocer en el Capítulo 3.

Seguidamente en el Capítulo 4 se enfoca en mencionar los distintos modelos de programación y optimizaciones basados en la modelo de investigación del proyecto *Estudio de rendimiento en GPU*, de Carlos Juega Reimúndez (2010).

En el Capítulo 5, se demostrara los resultados y comparaciones de rendimientos teniendo en cuenta las optimizaciones que favorecen a la implementación de CUDA, así como también se dará a conocer los beneficios que se obtienen con la implementación de CUDA como lo menciona la investigación que tiene por nombre *Implementación CPU-GPU,* de Misael Angeles Arce, Georgina Flores Becerra, and Antonio M. Vidal del Instituto Tecnológico de Puebla.

CAPÍTULO III

3.1. MARCO METODOLÓGICO

Como se explicó en el Capítulo 1, este proyecto pretende presentar un estudio de rendimiento utilizando la aceleración por GPU. Para el estudio que se realizará en este proyecto, serán basados mediante una investigación descriptiva por medio de antecedentes de diversos autores sobre el mismo tema propuesto. Así mismo se busca estudiar y evaluar la influencia del rendimiento por causa de la aceleración de GPU.

Es por ello que es conveniente obtener previamente un modelo de rendimiento de la GPU. Dicho modelo debe ser capaz de predecir el impacto que tienen sobre el rendimiento distintas transformaciones de alto nivel.

Debido a que el objetivo de este trabajo de grado es llevar a cabo resultados favorables en cuanto al rendimiento del computador por medio de la GPU, la misma favorecerá a cualquier empresa u oficina que requiera trabajar con el procesamiento de cargas graficas en el computador para tener una optimización y fluidez en el área de trabajo.

3.2. DISEÑO DE INVESTIGACIÓN

Se pretende utilizar el diseño de investigación no experimental descriptiva ya que el objetivo principal es estudiar y observar el compartimento del rendimiento y los valores en lo que se manifiestan las diferentes variables sin ser manipuladas, para así analizar, desarrollar y obtener resultados favorables dentro del enfoque cuantitativo.

El análisis y desarrollo se realizara de manera natural es decir, observar las variables es su contexto natural.

Para justificar este trabajo, en la Figura 28 se muestra una gráfica comparativa de rendimiento entre tecnologías CPU y GPU, Seguidamente en la figura 32 se

muestran resultados favorables con respecto al Procesamiento de imágenes en una GPU vs CPU.

La Figura 38 corresponde a la multiplicación de matrices, algoritmo que servirá de ejemplo conductor del trabajo. En la gráfica se muestra la variación del tiempo de ejecución frente a distintas configuraciones de una implementación del algoritmo (número de bloques, hilos por bloque, etc.), pero manteniendo constante el número de recursos de los multiprocesadores.

Estudiaremos los fundamentos más básicos de la computación gráfica como paso previo para evaluar las características de las arquitecturas orientadas al procesamiento gráfico.

La programación paralela consiste en dividir un problema en sub-problemas, con el fin de optimizar el tiempo que tarda el mismo problema en ser resuelto como un todo. Para tal propósito fueron creadas las arquitecturas paralelas. Existen varias formas de clasificar el procesamiento paralelo. Puede considerarse a partir de la organización interna de los procesadores o desde el flujo de información a través del sistema. A continuación daremos a conocer las siguientes variables.

3.3. MÉTRICAS DE DESEMPEÑO

Los algoritmos secuenciales son evaluados en términos de su tiempo de ejecución en función del tamaño del problema. Se define el tamaño del problema como el número de datos involucrados en una ejecución.

Cuando se trata de algoritmos paralelos el tiempo de ejecución depende no solamente del tamaño del problema sino también de la arquitectura y el número de procesadores de la(s) computadora(s). Por lo tanto para evaluar el rendimiento de un algoritmo dado, es necesario usar métricas de desempeño como:

- Tiempos de ejecución
- Aceleración

3.3.1. TIEMPOS DE EJECUCIÓN

El tiempo de ejecución secuencial de un programa, es el tiempo transcurrido entre el inicio y el fin de la ejecución del mismo en una computadora secuencial. El tiempo de ejecución paralelo de un programa, es el tiempo transcurrido entre el momento en que comienza el cómputo paralelo y el momento en el que el último procesador termina su trabajo. Se denota como Ts al tiempo de ejecución secuencial y como Tp al tiempo de ejecución paralelo.

3.3.2. ACELERACIÓN

Cuando se evalúan algoritmos paralelos el principal interés es conocer qué tan eficiente es con respecto a la implementación en forma secuencial. La aceleración es la métrica que captura el beneficio relativo al resolver un problema de forma paralela, entonces la aceleración, denotada con S, se define como el cociente del tiempo que se tarda en completar el cómputo de la tarea usando un sólo procesador entre el tiempo que necesita para realizarlo con p procesadores trabajando en paralelo. Se asume que los p procesadores usados por el algoritmo paralelo deben ser idénticos al procesador usado por el algoritmo secuencial.

$$S = \frac{T_s}{T_p}.$$

Es importante mencionar que al comparar el rendimiento de un algoritmo (secuencial vs paralelo) se debe considerar que puede estar disponible más de una versión, no siendo todas adecuadas para el paralelismo. Al usar una computadora secuencial se espera que el algoritmo resuelva el problema en la menor cantidad de tiempo posible. Entonces se debe comparar el rendimiento del algoritmo paralelo contra el rendimiento del algoritmo secuencial más rápido que resuelve el mismo problema. Para este documento las bibliotecas secuenciales utilizadas (BLAS y LAPACK) están optimizadas, por lo tanto los algoritmos en los que se basan son los más rápidos.

3.4. TIPOS DE ARQUITECTURAS

3.4.1. ARQUITECTURAS PARALELAS

Michael J. Flynn propuso en 1966, una taxonomía (considerando sistemas con uno o varios procesadores) que se basa en el flujo que siguen los datos dentro de la máquina y de las instrucciones sobre esos datos. Tomando en cuenta que la operación normal de una computadora es recuperar instrucciones de la memoria y ejecutarlas en el procesador, se define como flujo de instrucciones a la secuencia de instrucciones leída de la memoria y como flujo de datos a las operaciones ejecutadas sobre los datos en el procesador. El procesamiento paralelo puede ocurrir en el flujo de instrucciones, en el flujo de datos o en ambos.

3.4.1.1. MECANISMOS DE CONTROL

De acuerdo a los mecanismos de control, las computadoras se clasifican en los siguientes tipos:

- **SINGLE INSTRUCTION STREAM, SINGLE DATA STREAM (SISD):**

Las computadoras de este tipo cuentan con una unidad de control, un procesador (PU) y una unidad de memoria, es decir, tienen un único flujo de instrucciones sobre un único flujo de datos, la ejecución de las instrucciones es secuencial (Figura 1).

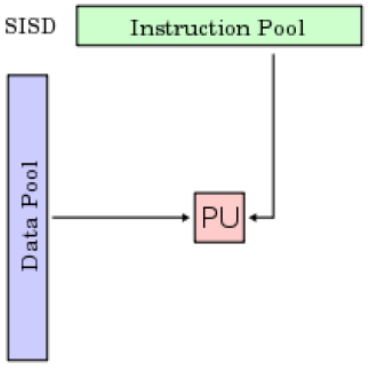


Figura 1: Arquitectura SISD

- **SINGLE INSTRUCTION STREAM, MULTIPLE DATA STREAM (SIMD):**

Estas computadoras tienen un único flujo de instrucciones que operan sobre múltiples flujos de datos, es decir, muchos procesadores (PU) bajo la supervisión de una unidad de control común. El procesamiento es síncrono, aunque la ejecución sigue siendo secuencial como en el caso anterior, todos los procesadores realizan la misma instrucción pero con diferentes conjuntos de datos. Por esta razón existirá concurrencia simulada, esta clasificación da origen a la máquina paralela (figura 2).

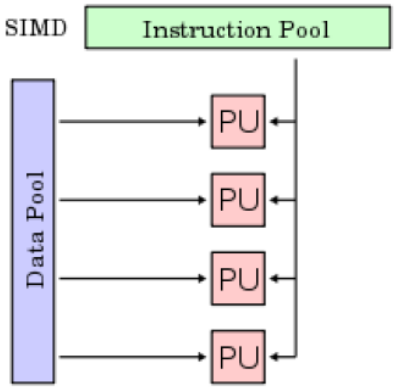


Figura 2: Arquitectura SIMD

- **MULTIPLE INSTRUCTION STREAM, SINGLE DATA STREAM (MISD):**

Las computadoras de este tipo (figura 3), cuentan con múltiples instrucciones que operan sobre un único flujo de datos, es decir, las instrucciones pasan a través de múltiples procesadores (PU). Estos sistemas operan de dos formas:

- Varias instrucciones operando simultáneamente sobre un único dato.
- Varias instrucciones operando sobre un dato que se va convirtiendo en un resultado que será la entrada para la siguiente etapa, todos los procesadores pueden trabajar de forma concurrente.

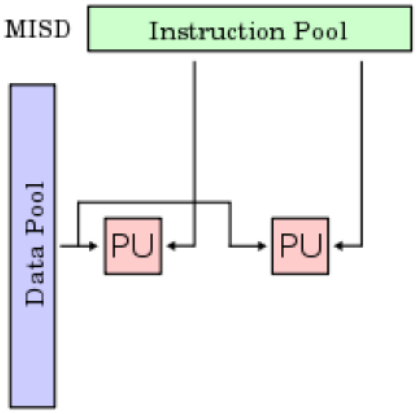


Figura 3: Arquitectura MISD

- **MULTIPLE INSTRUCTION STREAM, MULTIPLE DATA STREAM (MIMD):**

Estas computadoras tienen flujo de múltiples instrucciones que operan sobre múltiples datos. Son máquinas con memoria compartida que permiten ejecutar varios procesos simultáneamente (sistemas multiprocesador). Cada procesador (PU) es capaz de ejecutar su programa con diferentes datos, esto significa que los procesadores operan asíncronamente, es decir, pueden estar haciendo diferentes cosas en diferentes datos al mismo tiempo (figura 4).

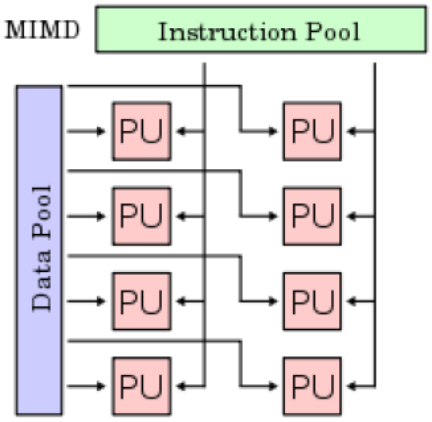


Figura 4: Arquitectura MIMD

3.4.1.2. ESPACIOS DE DIRECCIONAMIENTO

Cuando se resuelve un problema entre varios procesadores es necesario que estos se comuniquen entre sí para intercambiar información. El paso de mensajes y la memoria compartida proveen dos formas d comunicación.

3.4.2. ARQUITECTURAS DEL SISTEMA DE LA GPU

3.4.2.1. ARQUITECTURA DEL SISTEMA HETEROGÉNEO CPU-GPU

David Patterson, afirma que la arquitectura de un sistema de computación heterogéneo formado por una CPU y una GPU puede describirse a alto nivel con dos características primarias: primero, cuantos subsistemas funcionales o chips se utilizan y cuáles son las tecnologías y las topologías de interconexión; segundo, que subsistema de memoria están disponibles para estos subsistemas funcionales.

A continuación se mostrara un diagrama de bloques de alto nivel de un PC clásico hacia 1990.

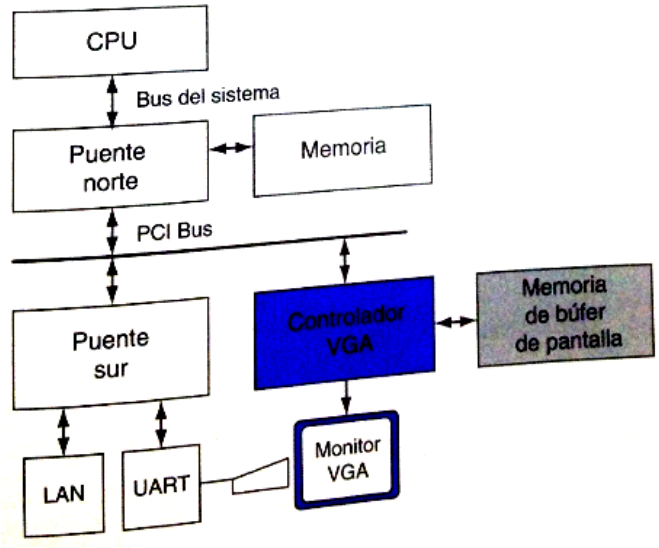


Figura 5: Controlador **VGA**, Se encarga de la visualización de los gráficos de la memoria de búfer de pantalla.

El puente norte contiene interfaces de alto ancho de banda que conectan la CPU, la memoria y el bus PCI. El puente sur consta de interfaces y dispositivos clásicos: bus ISA (audio, LAN), controlador de interrupciones controlador de DMA y temporizador. En este sistema, la visualización está controlada por un subsistema de búfer de pantalla, adjunto al bus PCI, llamado VGA (Video Graphics Array). Los subsistemas gráficos con elementos del procesamiento (GPU) no existían en el panorama de los PC de 1990.

La las siguientes dos figuras 6, se ilustra dos configuraciones todavía en uso actualmente. Se caracterizan por disponer de GPU y CPU separadas, en chips distintos con sus respectivos subsistemas de memoria. En la figura 6 (a), con una CPU Intel, la GPU está conectada con una conexión PCI Express 2.0 de 16 canales con ritmo de transferencia pico de 16 GB/s (pico de 8 GB/s cada dirección). De forma similar en la figura 6 (b), con una CPU AMD, la GPU está asociada al conjunto de chips, también con una conexión PCI Express del mismo ancho de banda. En ambos casos, la GPU puede acceder a la memoria de CPU y viceversa, aunque con un ancho de banda menor que el acceso a su memoria. En el sistema AMD, el puente norte o controlador de memoria, está integrado con la CPU.

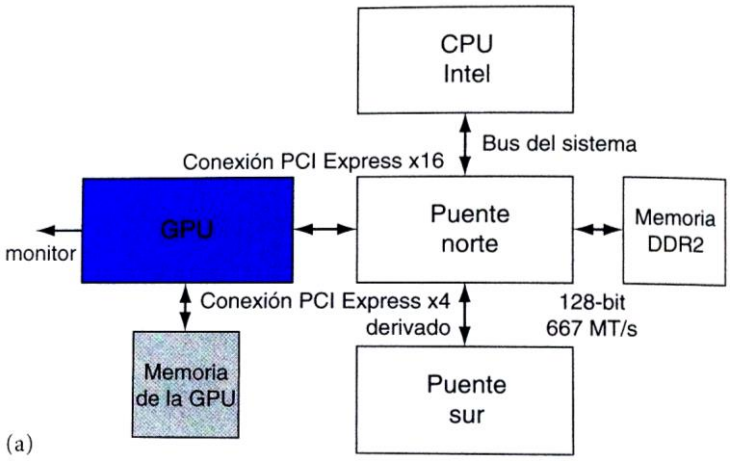


Figura 6 (a): Pc Actual con CPU Intel

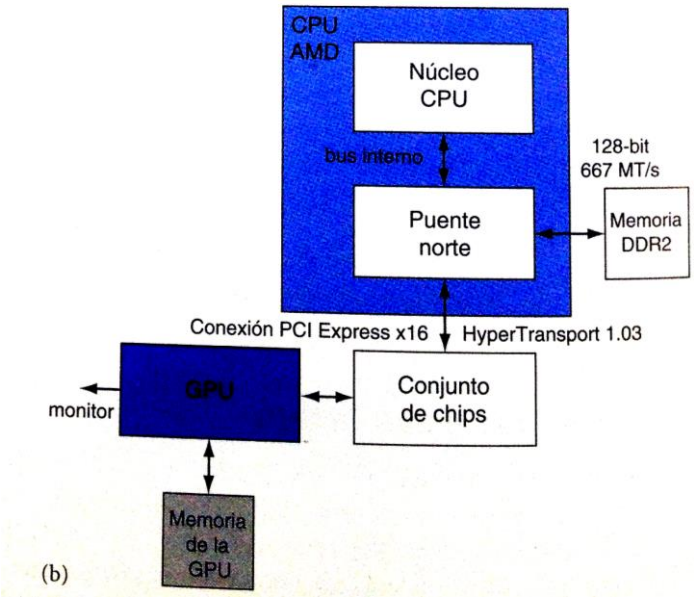


Figura 6 (b): Pc Actual con CPU AMD

3.4.2.2. ARQUITECTURA DE MEMORIA GPU UNIFICADA (UMA):

Un sistema de arquitectura de memoria unificada (UMA) es una variación de bajos costes de estos sistemas. Se unifica el procesamiento del sombreado de vértices, geometría y pixeles con la computación paralela en el mismo procesador, a diferencia de las GPUs anteriores que tenían procesadores separados y especializado para cada tipo de procesamiento. Utiliza solo la memoria de la CPU, eliminando la memoria GPU. Incorporan GPUs de prestaciones bajas, puesto que sus prestaciones están limitadas por el ancho de banda del sistema de memoria y la latencia de acceso de memoria, mientras que la memoria de la GPU proporciona un elevado ancho de banda y baja latencia.

El conjunto de procesadores programable está altamente integrado con procesadores de propósito específico (o función fija) para filtrado de texturas, rasterización, operaciones de raster, anti-aliasing, compresión, descompresión, visualización, decodificaciones de video y procesamiento de video de alta definición. Si las comparamos con las CPUs multinucleo, las GPUs con muchos núcleos tienen un punto de partida diferente para el diseño de la arquitectura, se centran en la

ejecución de muchos hilos paralelos en muchos núcleos. Como consecuencias de la utilización de muchos núcleos más sencillos y de las optimizaciones para aprovechar el paralelismo de datos entre los grupos de hilos, la mayor parte de los transistores del chip están delicados a cálculo en detrimento de las caches integradas en el mismo chip y otros componentes.

3.4.2.3. SISTEMA MULTI-GPU SLI

Una variación de altas prestaciones es un sistema con varias GPUs, normalmente dos o cuatros, trabajando en paralelo, con sus monitores conectados en cadena (*Daisy-chain*). Un ejemplo es el sistema de multi-GPU SLI (Scalable link interconnect) de NVIDIA, diseñado para juegos y estaciones de trabajo de altas prestaciones.

Actualmente lo sistema integran la GPU con el puente norte (Intel) o el conjunto de chips (AMD) con o sin memoria gráfica.

3.4.3. ARQUITECTURA DEL PIPELINE

Tradicionalmente, los procesadores gráficos funcionan mediante un pipeline de procesamiento formado por etapas muy especializadas en las funciones que desarrollan y que se ejecutan en un orden preestablecido. Cada una de las etapas del pipeline recibe la salida de la etapa anterior y proporciona su salida a la etapa siguiente. Debido a la implementación mediante una estructura de pipeline, el procesador gráfico puede ejecutar varias operaciones en paralelo. Como este pipeline es específico para la gestión de gráficos, normalmente se denomina pipeline gráfico o pipeline de renderización. La operación de renderización consiste en proyectar una representación en tres dimensiones de una imagen en dos dimensiones (que es el objetivo de un procesador gráfico).

La entrada del procesador gráfico es una secuencia de vértices agrupados en primitivas geométricas (polígonos, líneas y puntos), que son tratadas secuencialmente

por medio de cuatro etapas básicas, tal como muestra el pipeline simplificado de la figura 7.

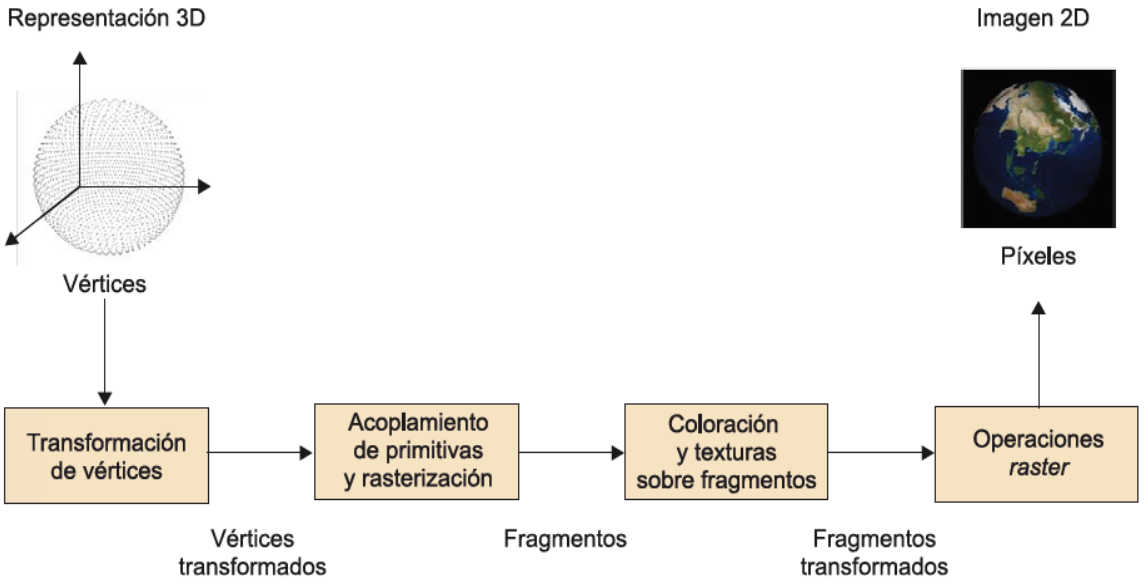


Figura 7: *Pipeline* simplificado de un procesador gráfico.

La etapa de transformación de vértices consiste en la ejecución de una secuencia de operaciones matemáticas sobre los vértices de entrada basándose en la representación 3D proporcionada. Algunas de estas operaciones son la actualización de la posición o la rotación de los objetos representados, la generación de coordenadas para poder aplicar texturas o la asignación de color a los vértices. La salida de esta fase es un conjunto de vértices actualizados, uno para cada vértice de entrada.

En la etapa de acoplamiento de primitivas y rasterización, los vértices transformados se agrupan en primitivas geométricas basándose en la información recibida junto con la secuencia inicial de vértices. Como resultado, se obtiene una secuencia de triángulos, líneas y puntos. Estos puntos son posteriormente procesados en una etapa llamada rasterización. La rasterización es un proceso que determina el conjunto de píxeles afectados por una primitiva determinada. Los resultados de la

rasterización son conjuntos de localizaciones de píxeles y conjuntos de fragmentos. Un fragmento tiene asociada una localización y también información relativa a su color y brillo o bien a coordenadas de textura. Como norma general, podemos decir que, de un conjunto de tres o más vértices, se obtiene un fragmento.

En la etapa de aplicación de texturas y coloreado, el conjunto de fragmentos es procesado mediante operaciones de interpolación (predecir valores a partir de la información conocida), operaciones matemáticas, de aplicación de texturas y de determinación del color final de cada fragmento. Como resultado de esta etapa, se obtiene un fragmento actualizado (coloreado) para cada fragmento de entrada.

En las últimas etapas del pipeline, se ejecutan operaciones llamadas raster, que se encargan de analizar los fragmentos mediante un conjunto de tests relacionados con aspectos gráficos. Estos tests determinan los valores finales que tomarán los píxeles. Si alguno de estos tests falla, se descarta el píxel correspondiente y, si todos son correctos, finalmente el píxel se escribe en la memoria (framebuffer). La figura 8 muestra las diferentes funcionalidades del pipeline gráfico

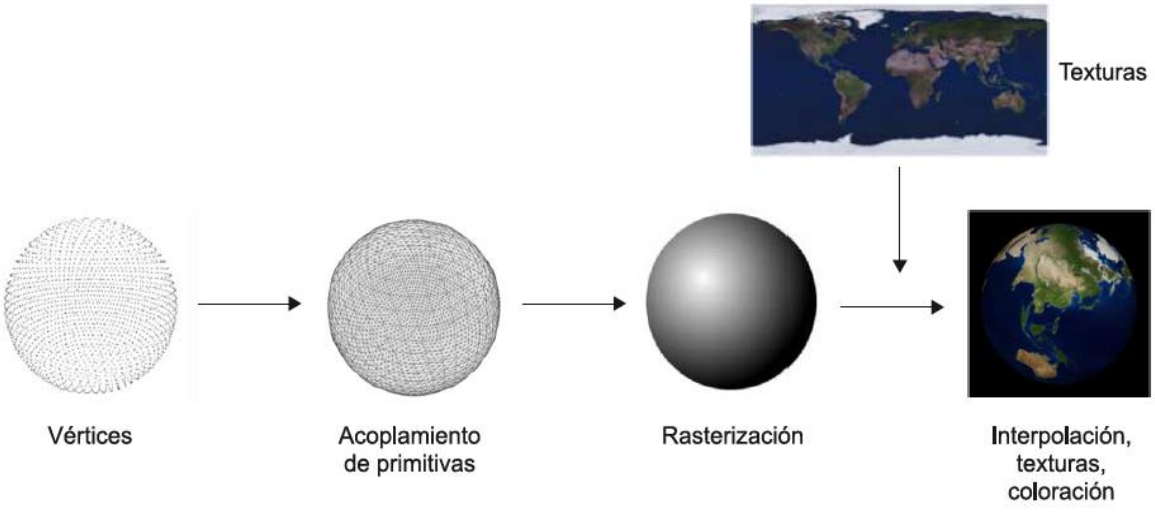


Figura 8: Resumen de las funcionalidades del *pipeline* gráfico

3.4.4. ARQUITECTURAS ORIENTADAS AL PROCESAMIENTO GRÁFICO

En este apartado, vamos a estudiar las motivaciones y los factores de éxito del desarrollo de arquitecturas basadas en computación gráfica, las características básicas de estas arquitecturas y el caso particular de la arquitectura Nvidia y AMD orientadas a gráficos como caso de uso. Finalmente, vamos a analizar las posibilidades y limitaciones para poder ser utilizadas para computación de propósito general.

3.4.4.1. ARQUITECTURA NVIDIA

Nvidia presentó a finales del 2006 una nueva línea de hardware orientado a la computación general de otras prestaciones llamada Tesla, que se empezó a desarrollar a mediados del 2002. Tesla ofrece un hardware de altas prestaciones (en forma, por ejemplo, de bloques de procesadores gráficos) sin ningún tipo de orientación a aplicaciones gráficas. A pesar de que no es la implementación de Tesla más actual, en este apartado nos vamos a centrar en la arquitectura G80, ya que representó un salto tecnológico importante por el hecho de implementar una arquitectura unificada y, tal como vamos a ver en el siguiente apartado, vino con el modelo de programación CUDA.

La arquitectura G80 de Nvidia se define como una arquitectura totalmente unificada, sin diferenciación a nivel de hardware entre las diferentes etapas que forman el pipeline gráfico, totalmente orientada a la ejecución masiva de flujos y que se ajusta al estándar IEEE 754. La figura 9 muestra el esquema completo de la arquitectura G80.

La aparición de la arquitectura G80 representó una mejora de la capacidad de procesamiento gráfico y un incremento de las prestaciones respecto a la generación anterior de GPU, pero la clave en orden al ámbito de la computación general fue la mejora de la capacidad de cálculo en coma flotante. También se añadieron al pipeline para cumplir las características definidas por Microsoft en DirectX 10.

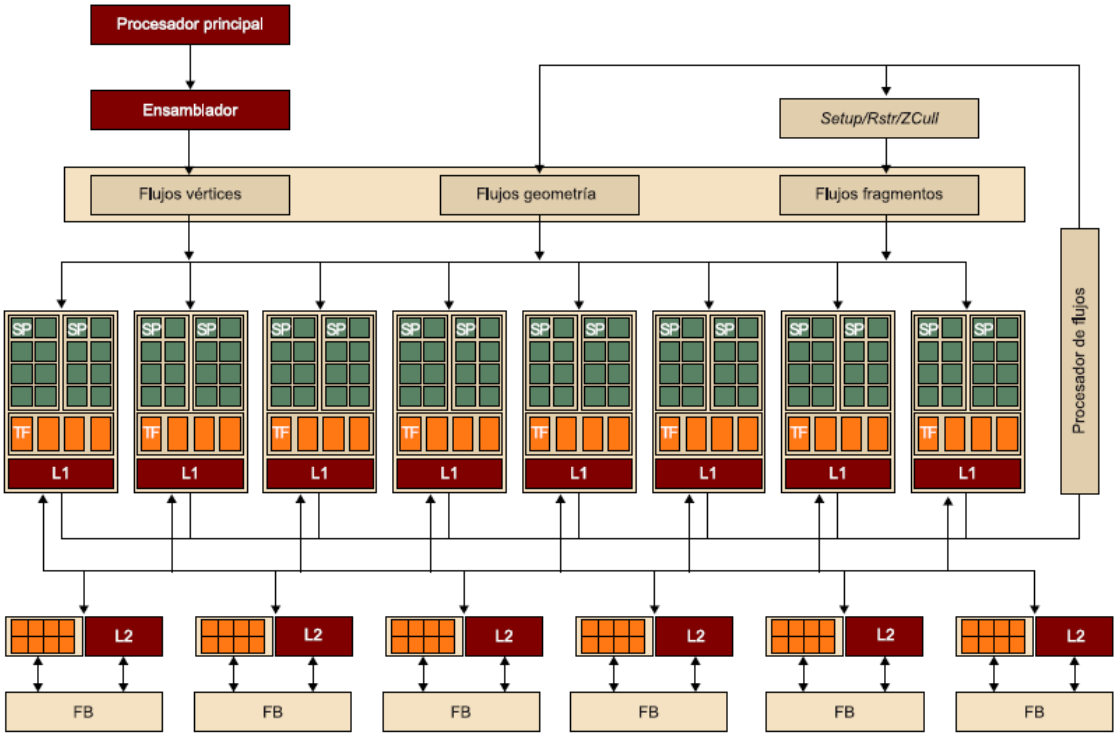


Figura 9: Arquitectura (unificada) de la serie G80 de Nvidia

3.4.4.2. ARQUITECTURA AMD (ATI)

De forma similar a la tecnología de Nvidia, la tecnología GPU desarrollada por AMD/ATI ha evolucionado muy rápidamente en la última década hacia la computación GPGPU. En este subapartado, vamos a estudiar las principales características de la arquitectura de la serie R600 de AMD (que es el equivalente a la Nvidia G80) y la arquitectura CU (implementada por la familia Evergreen de AMD), que es un desarrollo nuevo y está asociada al modelo de programación OpenCL.

La serie R600 de AMD implementa una arquitectura unificada como en el caso de la Nvidia G80. También incluye un shader de geometría que permite ejecutar operaciones de creación de geometría en la GPU y así no tener que sobrecargar la CPU. La serie R600 incorpora 320 SP, que son muchos más que los 128 de la G80. Sin embargo, esto no quiere decir que sea mucho más potente, ya que los SP de las

dos arquitecturas funcionan de forma bastante diferente. La figura 10 muestra el esquema de la arquitectura de la serie R600 de AMD.

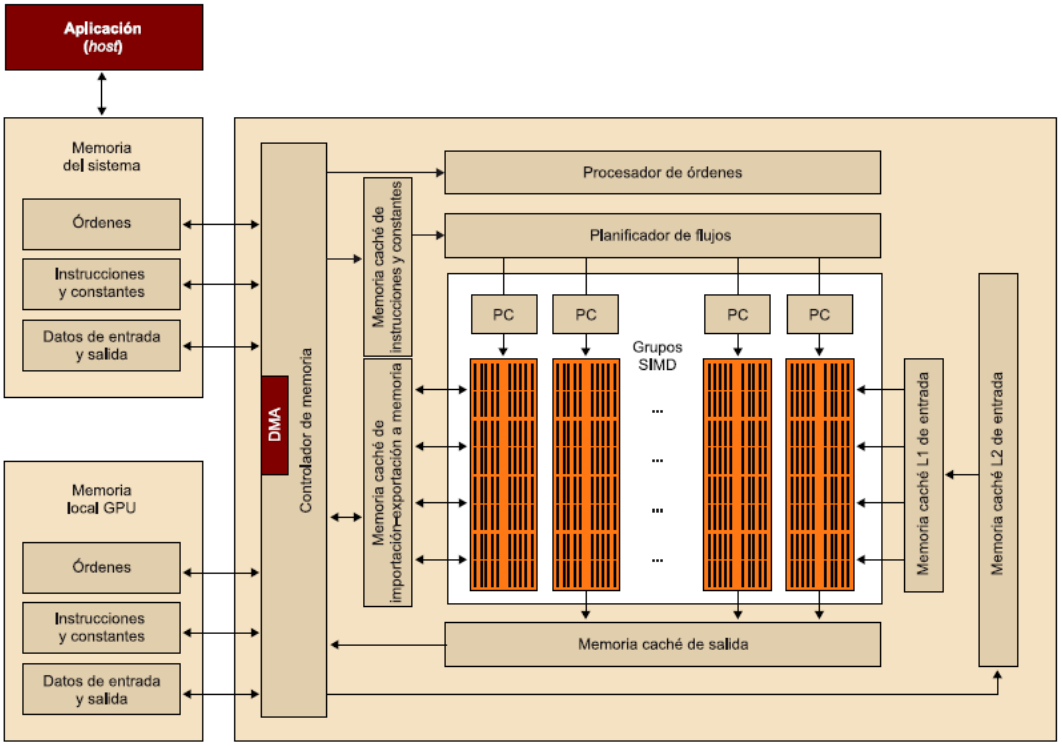


Figura 10: Diagrama de bloques de la arquitectura de la serie R600 de AMD

3.4.4.3. ARQUITECTURA CUDA:

CUDA es un modelo de programación paralela mediante el uso de dispositivos de aceleración gráfica (GPU) y un entorno de desarrollo diseñado para superar este desafío manteniendo una baja curva de aprendizaje para los programadores que están familiarizados con el lenguaje de programación estándar C.

En su esencia CUDA tiene tres elementos clave: El primero es una jerarquía de grupos de hilos, el segundo es un modelo de recursos compartidos y el tercero es un grupo de instrucciones de sincronización, que son expuestos al programador como un Conjunto mínimo de extensiones de C.

Estos elementos orientan al programador a dividir el problema en sub-problemas secundarios que pueden ser resueltos de forma paralela y después en subprogramas más pequeños que pueden ser resueltos en forma cooperativa en paralelo. Esto permite una escalabilidad transparente, ya que cada subprograma puede ser programado para ser resuelto en cualquiera de los núcleos disponibles del procesador. Un programa compilado con CUDA se puede ejecutar en cualquier número de núcleos del procesador y el sistema de ejecución solo necesita saber el número de procesadores físicos.

La arquitectura de las tarjetas NVIDIA aptas para CUDA corresponden a un multiprocesador, como se ilustra en la figura 11. En esta figura puede apreciarse la arquitectura de una tarjeta NVIDIA con 8 multiprocesadores, con 16 núcleos cada uno y sus respectivas unidades de memoria.

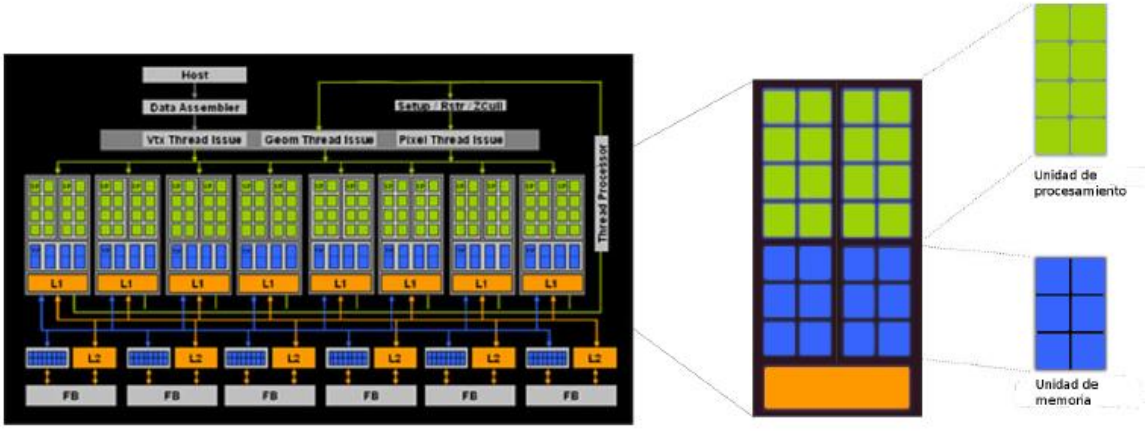


Figura 11: Arquitectura de Tarjeta NVIDIA con 128 núcleos

De acuerdo con lo visto en la sección 3.4.1.1 el mecanismo de control de la arquitectura CUDA corresponde al de las computadoras tipo SIMD, ya que la programación en paralelo en CUDA consiste en ejecutar un conjunto de hilos que realizan las mismas operaciones sobre múltiples datos.

Por otra parte, CUDA usa la memoria compartida como medio de comunicación entre los multiprocesadores (módulos L1 y L2 de la figura 11) que componen una tarjeta NVIDIA apta para CUDA.

De acuerdo a la teoría de las arquitecturas paralelas, las tarjetas NVIDIA se clasifican como mulitprocesadores, tomando en cuenta que una tarjeta NVIDIA está integrada por un conjunto de procesadores, entonces la granularidad de éstas es de grano fino.

3.4.4.4. ARQUITECTURA DE OPENCL

OpenCL es una interfaz estándar, abierta, libre y multiplataforma para la programación paralela. La principal motivación para el desarrollo de OpenCL fue la necesidad de simplificar la tarea de programación portátil y eficiente de la creciente cantidad de plataformas heterogéneas, como CPU multinúcleo, GPU o incluso sistemas incrustados. OpenCL fue concebida por Apple, a pesar de que la acabó desarrollando el grupo Khronos, que es el mismo que impulsó OpenGL y su responsable.

OpenCL consiste en tres partes: la especificación de un lenguaje multiplataforma, una interfaz a nivel de entorno de computación y una interfaz para coordinar la computación paralela entre procesadores heterogéneos. OpenCL utiliza un subconjunto de C99 con extensiones para el paralelismo y utiliza el estándar de representación numérica IEEE 754 para garantizar la interoperabilidad entre plataformas.

Existen muchas similitudes entre OpenCL y CUDA, aunque OpenCL tiene un modelo de gestión de recursos más complejo, ya que soporta múltiples plataformas y portabilidad entre diferentes fabricantes. OpenCL soporta modelos de paralelismo tanto a nivel de datos como a nivel de tareas. En este subapartado, nos vamos a centrar en el modelo de paralelismo a nivel de datos, que es equivalente al de CUDA.

3.5. DIFERENCIAS, SIMILITUDES E INTERCONEXIÓN ENTRE LAS ARQUITECTURAS CPU Y GPU.

Poder contar con una arquitectura de trabajo híbrida permite aprovechar las ventajas de la CPU y las GPU. A continuación mostraremos un cuadro comparativo en donde se mostraras las Diferencias y Similitudes entre ambas arquitecturas CPU-GPU.

CPU	GPU
• Su Arquitecturas es Multicore	• Su Arquitecturas es Many-Cores
• Provee mejor desempeño a soluciones secuenciales	• Se centran en optimizar el desempeño de las aplicaciones
• Provee lógica de control compleja para la ejecución paralela de código secuencial	• Se obtienen mejoras en la velocidad del performance.
• Incluye memorias caché más rápidas y más grandes para disminuir la latencia de las instrucciones.	• Las memorias caché son pequeñas
• Optimiza el ancho de banda para atender a todas las aplicaciones, operaciones de entrada/salida y funciones del sistema operativo.	• Logra un mayor ancho de banda diez veces superior al de las CPU contemporáneas.
• Alto rendimiento sobre un único hilo.	• Alto rendimiento cuando se ejecutan tareas paralelas.
	• Se implementa CUDA
	• OpenGL

Tal como se ha indicado, la filosofía de diseño de las GPU está influida por la industria del videojuego, que ejerce una gran presión económica para mejorar la capacidad de realizar una gran cantidad de cálculos en coma flotante para procesamiento gráfico. Esta demanda hace que los fabricantes de GPU busquen formas de maximizar el área del chip y la cantidad de energía dedicada a los cálculos en coma flotante. Para optimizar el rendimiento de este tipo de cálculos, se ha optado por explotar un número masivo de flujos de ejecución. La estrategia consiste en explotar estos flujos de tal manera que, mientras que unos están en espera para el acceso a memoria, el resto pueda seguir ejecutando una tarea pendiente.

Tal como se muestra en la figura 11.5, en este modelo se requiere menos lógica de control para cada flujo de ejecución. Al mismo tiempo, se dispone de una pequeña memoria caché que permite que flujos que comparten memoria tengan el ancho de banda suficiente para no tener que ir todos a la DRAM. En consecuencia, mucha más área del chip se dedica al procesamiento de datos en coma flotante.

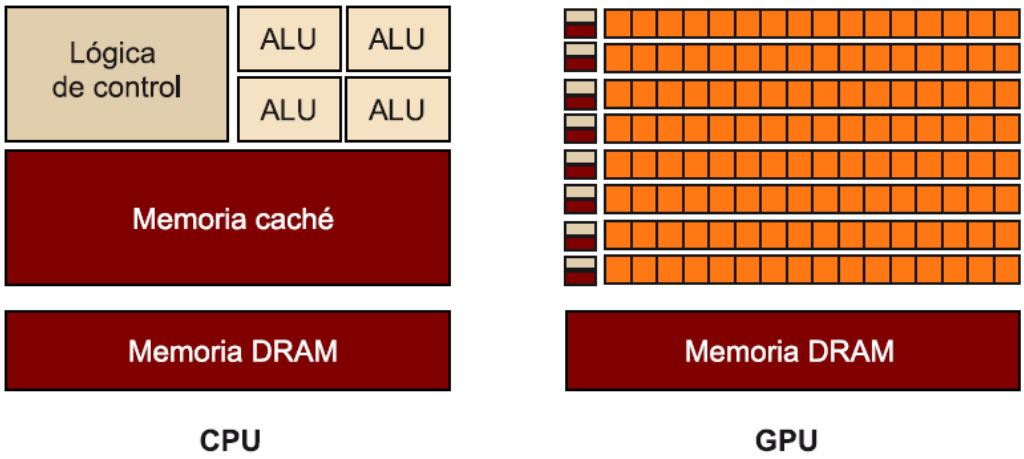


Figura 11.5: Comparativa de la superficie dedicada típicamente a computación, memoria y lógica de control para CPU y GPU

Tal como veremos en detalle más adelante, además de disponer de muchos flujos de ejecución dispuestos en núcleos más sencillos que los de las CPU, otras características básicas de las arquitecturas GPU son las siguientes:

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

En un sistema que dispone de una o de múltiples GPU, normalmente las GPU son vistas como dispositivos externos a la CPU (que puede ser multinucleo o incluso un multiprocesador), que se encuentra en la placa base del computador, tal como muestra la figura 11.6. La comunicación entre CPU y GPU se lleva a cabo por medio de un puerto dedicado. En la actualidad, el PCI Express o PCIe (peripheral component interconnect express) es el estándar para ejecutar esta comunicación.

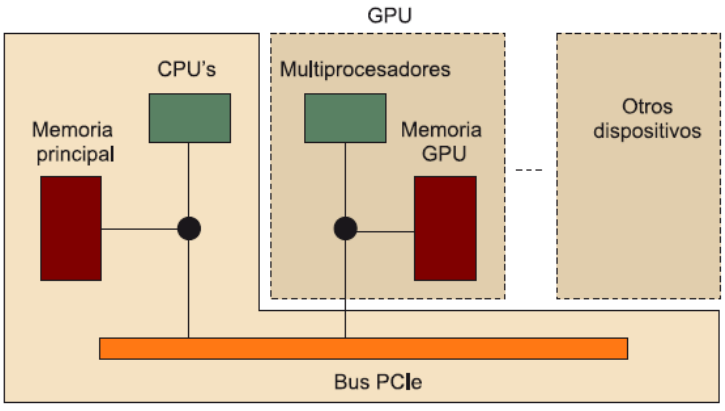


Figura 11.6: Interconexión entre CPU y GPU mediante PCIe

3.5.1. CPU vs GPU

Para la programación en paralelo sobre GPUs se implementan programas en C con extensiones para CUDA. Algunas secciones de código de estos programas se ejecutan en CPU (las secciones de código secuencial), mientras que aquellas secciones de código paralelo se ejecutan en GPUs en forma de hilos, como se verá en los siguientes apartados.

3.5.2. DEFINICIÓN DE HOST Y DEVICE

Es importante conocer las diferencias entre el host y el device para que el rendimiento de los programas CUDA que se escriban sea eficiente. Es importante conocer las diferencias entre el **host** y el **device** para que el rendimiento de los programas CUDA que se escriban sea eficiente.

3.5.3. Hilos y RAM

Hilos: Los sistemas host pueden soportar un número limitado de hilos concurrentes. Por ejemplo, los servidores con cuatro procesadores (quad-core) pueden ejecutar solamente 16 hilos en paralelo. En contraste, la unidad paralela ejecutable más pequeña sobre un dispositivo (llamada warp o urdimbre2) comprende 32 hilos. Por ejemplo todas las GPUs de la tarjeta NVIDIA pueden soportar 768 hilos activos por multiprocesador. Los hilos de la CPU son generalmente entidades pesadas, el sistema operativo tiene que estar encendiendo y apagando hilos para proveer ejecución multi-hilo, es muy costoso pasar de un estado a otro. Por otro lado los hilos de la GPU son muy ligeros.

RAM: Ambos dispositivos (host y device) tienen su propia RAM. En el host generalmente la RAM es igualmente accesible para todo el código (con limitaciones establecidas por el sistema operativo). En el device la RAM es dividida virtual y físicamente en diferentes tipos, cada una de ellas para satisfacer diferentes necesidades.

CAPÍTULO IV

PROGRAMACION Y OPTIMIZACIONES

4.1. PROGRAMACIÓN DE LAS GPU

La programación de las GPUs multiprocesador es cualitativamente diferente de la programación de otros multiprocesadores como las CPU multinucleo. El número de hilos y el paralelismo a nivel de datos de la GPU es don o tres órdenes de magnitud mayor que en las CPUs, escandalo en 2008 a cientos de núcleos y decenas de miles de hilos concurrentes. Las GPUs siguen aumentando su paralelismo, doblándolo cada 12 o 18 meses impulsados por el aumento de la densidad de integración de los circuitos de acuerdo con la ley de Moore (1965) y las mejoras de la eficiencia de la arquitectura.

4.1.1. CÁLCULO PARALELO EN GPU:

Como anteriormente en el capítulo 3 lo hemos mencionado, CUDA (*Compute Unified Device Architecture*) es un framework de desarrollo de aplicaciones paralelas de propósito general mediante el uso de dispositivos de aceleración gráfica (GPUs). Se trata de la alternativa desarrollada por NVIDIA en el campo de la GPU y está compuesta por un nuevo modelo de programación que permite a los desarrolladores el acceso a los dispositivos con capacidades CUDA, así como un conjunto de herramientas y librerías de desarrollo para facilitar la labor de los mismos en la programación y depuración de aplicaciones bajo este modelo.

En la Figura 12 podemos ver un esquema general del conjunto de elementos de la capa software de CUDA. Como vemos, la capa más baja comprende el propio motor de ejecución situado dentro de los dispositivos gráficos, seguido del soporte proporcionado por el kernel del sistema operativo. Sobre éste se sitúa por un lado el soporte DirectX para hacer uso directo de las llamadas del kernel, y por otro el driver

CUDA que puede ser utilizado directamente o mediante una nueva capa de abstracción, como es el caso de OpenCL.

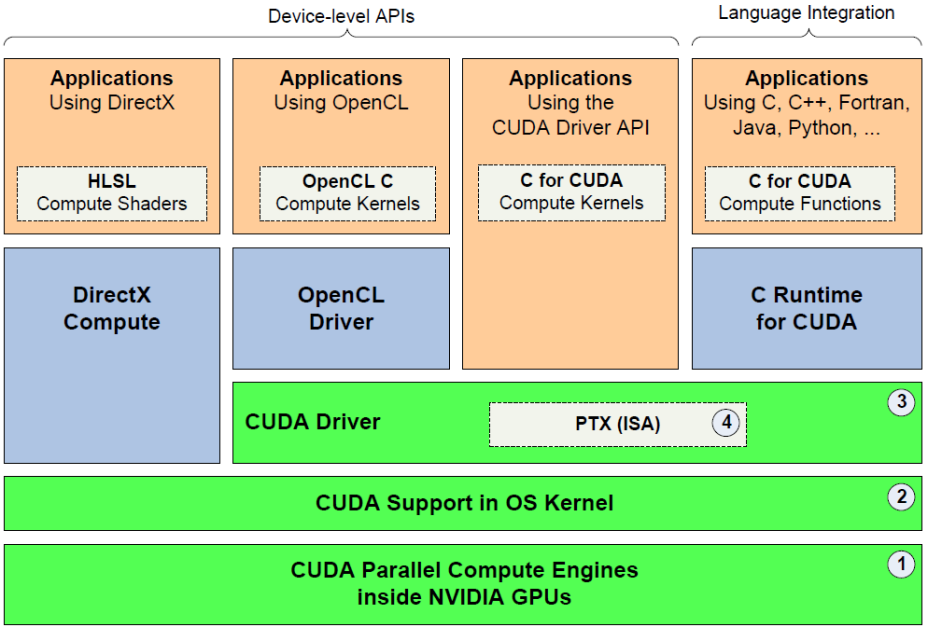


Figura 12: Elementos de la capa software en el modelo CUDA

4.2. MODELO DE PROGRAMACIÓN CUDA

El modelo de programación CUDA asume que los hilos CUDA se ejecutan en una unidad física distinta que actúa como coprocesador (device) al procesador (host) donde se ejecuta el programa (Figura 13). CUDA C es una extensión del lenguaje de programación C, que permite al programador definir funciones C, llamadas kernels, que, al ser llamadas, se ejecutan en paralelo por N hilos diferentes. Los kernels se ejecutan de forma secuencial en el device.

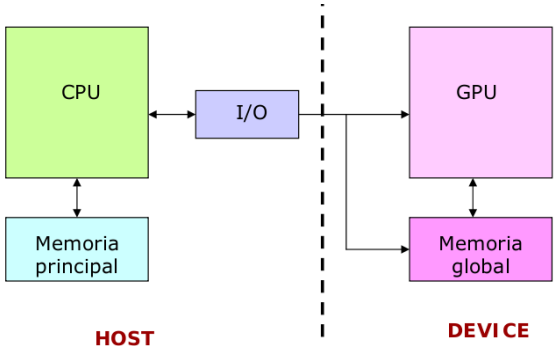


Figura 13: Arquitectura CPU-GPU

Como ejemplo, el siguiente código muestra cómo se define un kernel y cómo se llaman desde un programa:

```
// Kernel definition
__global__ void VecAdd (f loat * A, f loat * B, f loat * C)
{
        int i = threadIdx . x ;
        C[ i ] = A[ i ] + B[ i ] ;
}
int main ( )
{
. . .
        // Kernel inv o c a t i on wi th N t h r e a d s
        VecAdd<<<1, N>>>(A, B, C) ;
}
```

Desde este punto de vista, la GPU se puede ver como un coprocesador de la CPU donde se ejecutarán las partes de código de una aplicación que sean intensivas en paralelismo de datos, en lo que puede considerarse un entorno de computación heterogénea. . Dado que no todo el código de una aplicación hará uso de este tipo de paralelismo, y los procesadores CUDA son inferiores en potencia a una CPU, se deberán utilizar ambas tecnologías para sacar el máximo partido al rendimiento de la aplicación. Así se diferencian dos partes en una aplicación que haga uso de CUDA: el código ejecutado en la CPU (host) y el ejecutado en la GPU (device).

Como dispositivo independiente de la jerarquía de memoria principal del sistema, la GPU necesitará de transferencias explícitas de datos entre ésta y su espacio de memoria, implicando esta acción una gran cantidad de tiempo adicional que afectará, en gran medida, al rendimiento de la aplicación.

Además de su correspondiente fase de ejecución, el host jugará un papel determinante en este entorno. Se verá implicado en la reserva y liberación de

memoria en el dispositivo, así como en la realización de las transferencias de datos entre ambos ámbitos, y determinará cuándo el dispositivo comenzará a ejecutar cada uno de sus kernel. Se puede ver un esquema de las distintas fases en la Figura 14.

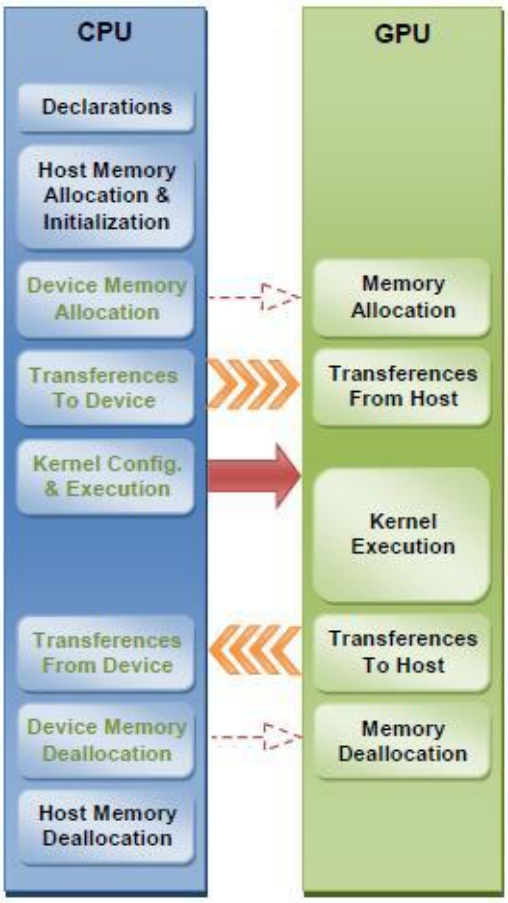


Figura 14: Fase de ejecución en CUDA

Entrando en un mayor nivel de detalle, la Figura 15 muestra un esquema del modelo de ejecución. Cada fase del código que está destinado a ejecutarse en una o varias GPUs recibe en nombre de kernel. Este código será ejecutado por todos y cada uno de los hilos del/los dispositivo/s, por lo que estará escrito en función del identificador de estos hilos, entre otros factores.

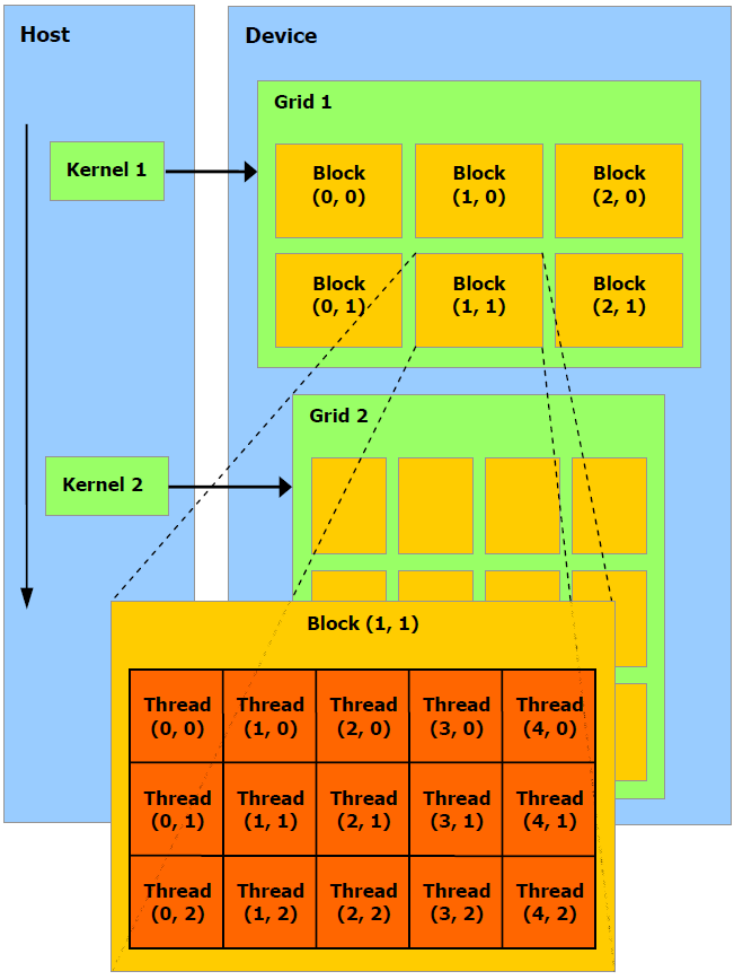


Figura 15: Modelo de ejecución en CUDA

Un bloque es una agrupación de hilos y viene a representar la unidad mínima de asignación entre dichos hilos y un multiprocesador del dispositivo. De esta forma, si el número de hilos por bloque es demasiado alto, tendremos menos bloques por multiprocesador ejecutándose en paralelo, y si es demasiado bajo, justo lo contrario. Existe un máximo de bloques e hilos que un multiprocesador puede manejar físicamente en paralelo. Para un aprovechamiento óptimo de los recursos, debemos tratar de que el número de hilos por bloque sea múltiplo del máximo de hilos que soporta cada multiprocesador, y sea lo suficientemente grande como para que los bloques puedan cubrir todos los recursos disponibles. Se recomienda que este valor

coincida con una potencia de dos no excesivamente pequeña. También debemos intentar disponer de un número de bloques elevado para ocultar tiempos de latencia, y permitir que los multiprocesadores sigan trabajando cuando un bloque queda a la espera por algún motivo. A ser posible, esta cifra debe ser mayor que el doble del número de multiprocesadores del dispositivo.

Los bloques e hilos de cada kernel vienen englobados en una estructura denominada Grid. Esta estructura representa la materialización de cada kernel en un dispositivo, y es en la cual el programador establece el número de hilos por bloque y número total de bloques con los que se lanzará dicho código a la GPU. Así, se puede ejecutar el mismo kernel con la misma o diferente configuración de hilos y bloques sin que sea necesaria ningún tipo de modificación de su código.

En definitiva, para aprovechar la potencia de estos dispositivos es necesario familiarizarse, al menos, con los conceptos de kernel, Grid, bloque e hilo, sin detenernos en demasiados detalles de bajo nivel. Desde este punto, se podrán desarrollar aplicaciones con un rendimiento más o menos aceptable, llevando a cabo una programación con granularidad de hilo.

4.3. MODELO DE MEMORIA

La memoria es uno de los aspectos de las GPUs donde se podrán encontrar más lagunas de información, quizás, debido a temas de confidencialidad o a que los detalles existentes son de tan bajo nivel que resulta imposible su inclusión detallada en los manuales. Sin embargo, el correcto uso de los diferentes tipos de memoria que un dispositivo brinda será el mejor aliado si se quiere conseguir un mayor rendimiento.

En la Figura 16, se muestran los seis tipos de recursos que los dispositivos ofrecen para almacenar información, y el ámbito de visibilidad de los mismos: registros (registers), memoria local (local memory), memoria compartida (shared memory), memoria constante (constant memory), memoria de texturas (texture

memory) y memoria global (global memory). Por defecto, la coherencia entre los recursos que requieren ser manipulados manualmente, en caso de ser necesaria, deberá ser mantenida por el programador. Este hecho debe ser aún más tenido en cuenta cuando nos referimos a la interacción entre la memoria del host y la memoria del dispositivo. No obstante, existen formas de que ambos compartan información de manera automática, como veremos más adelante.

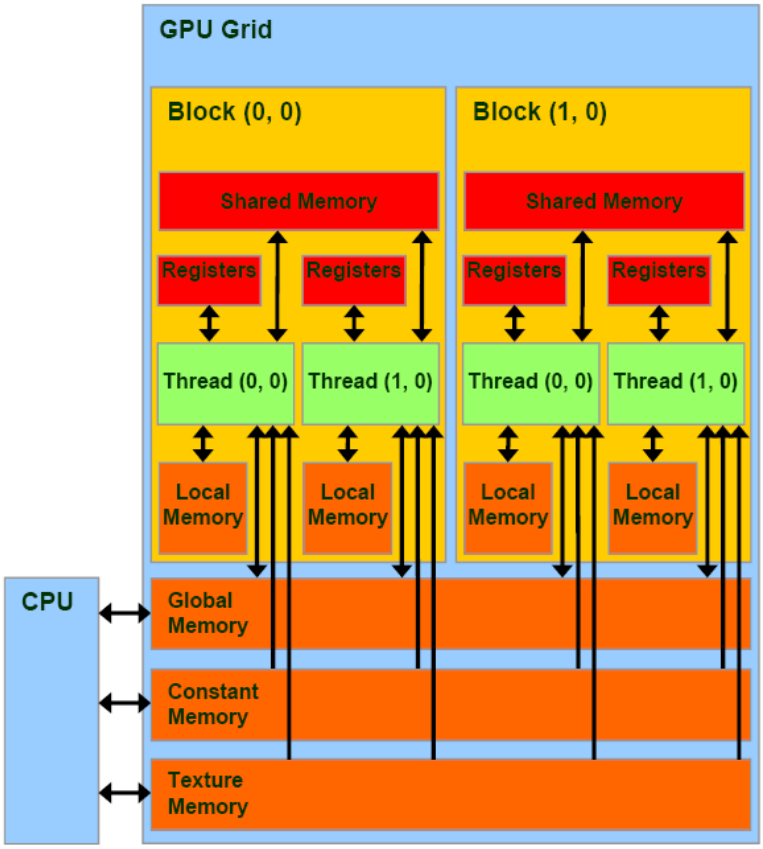


Figura 16: Modelo de memoria en CUDA

4.3.1. MEMORIA GLOBAL

La memoria global es la memoria más abundante, pero también la que se caracteriza por una mayor latencia y menor ancho de banda.

En CPU, sería equivalente a la memoria RAM. Se encuentra fuera del chip de procesamiento y es el medio principal para comunicar datos entre el host y el

dispositivo. Esta comunicación, por defecto, debe ser manualmente especificada por el programador. El tiempo de vida de los datos es de todo el programa, y estos son visibles por cualquier hilo de cualquier bloque que se encuentre en ejecución en el dispositivo.

Los hilos acceden a ella en grupos de half-warps. Para obtener el mejor rendimiento, los accesos deben ser coalescentes. Este concepto ha ido evolucionando a la vez que lo han hecho las GPUs de propósito general, siendo cada vez menos restrictivas. Para dispositivos con capacidad de cómputo 1.0 y 1.1, un acceso será coalescente cuando en cada half-warp:

1. El tamaño de palabra accedido por hilo sea de 4, 8 o 16 bytes.

2. El primer elemento de cada half-warp debe estar alineado a un múltiplo de 16 veces su tamaño, para que todos los elementos se encuentren dentro del mismo segmento de memoria.

3. Los hilos deben acceder a los datos secuencialmente: el hilo k debe acceder a la palabra k. Pueden existir hilos que no soliciten acceso a ningún dato, siempre que esto no altere el orden para los hilos siguientes.

Si alguna de las restricciones no se cumple, la penalización será máxima, y se llevarán a cabo accesos secuenciales independientes de 32 bytes por cada hilo del half-warp.

4.3.2. MEMORIA LOCAL

La memoria local es una manera alternativa de utilizar memoria global, acotando la visibilidad y el tiempo de vida de los datos a nivel de hilo y de kernel respectivamente. No es otro tipo diferente de memoria y se puede asumir todo lo dicho anteriormente en lo que a características se refiere, con algunas salvedades. Normalmente, esta memoria es utilizada como área de desbordamiento de los registros o para almacenar estructuras de datos de varios elementos, locales a cada

hilo. No es una decisión directa del programador el usarla o no, sino que es el compilador (estructuras de datos) y el runtime los que arbitrarán a su antojo su utilización. Esto supone que el hecho de realizar los accesos coalescentes no esté en nuestra mano, por lo que serán estas entidades las encargadas de situar los datos en memoria global, de manera que se obtengan los mejores resultados de acceso.

4.3.3. REGISTROS

Los registros son el recurso de almacenamiento más rápido de las GPUs, junto con la memoria compartida. Se encuentran dentro del chip de procesamiento y su visibilidad y tiempo de vida son similares a los de la memoria local. Como ya se ha comentado, al tratarse de un recurso limitado, cuando no quedan registros disponibles se utiliza la memoria local como área de desbordamiento. Al igual que ésta, el programador no puede controlar directamente su uso, aunque sí establecer algunas limitaciones.

4.3.4. MEMORIA COMPARTIDA

La memoria compartida es uno de los recursos más valiosos del dispositivo. Se encuentra dentro del chip y, a diferencia de los registros y la memoria local, su visibilidad es a nivel de bloque. Esto quiere decir que todos los hilos del mismo bloque pueden compartir información utilizando este medio. Es un tipo de memoria bastante limitado, y será el programador el encargado de mover los datos entre esta y el resto de recursos manualmente. Además, su tiempo de vida es a nivel de kernel y su latencia puede equipararse a la de los registros, en el mejor de los casos. Sin embargo, esto no siempre será así. Los accesos a la memoria se llevan a cabo en grupos de half-warps, por lo que se dispone de 16 bancos con tamaño de palabra de 4 bytes. Cuando varios hilos, pertenecientes al mismo half-warp, intenten acceder al mismo banco, ya sea para alcanzar el mismo elemento o elementos diferentes, dichos accesos serán atendidos en serie, y por tanto, la latencia será mayor.

4.3.5. MEMORIA CONSTANTE

La memoria constante es un tipo de memoria dentro del dispositivo, cuya función es ofrecer un nivel de caché por encima de la memoria global. Como su nombre indica, se trata de una memoria de sólo lectura a nivel de GPU, por lo que los datos deberán ser transferidos/asociados a ella desde el host, antes de proceder con la ejecución de un kernel que acceda a esos elementos. Su visibilidad y tiempo de vida son similares a los de la memoria global, pero hay que destacar que no guarda coherencia con la misma.

Actualmente, los dispositivos de NVIDIA cuentan con una memoria constante total de 64 KB. Este tamaño podría resultar interesante si se contempla desde el punto de vista de una memoria caché convencional de una CPU, donde es posible trabajar con un conjunto de datos de mayor tamaño al de su capacidad total, de manera transparente al usuario. Sin embargo, en este caso se encuentra la limitación de no poder utilizar ni un solo byte por encima de la capacidad de esta memoria, al menos de manera automática, por lo que el tamaño podría resultar algo escaso para objetivos de carácter general.

4.3.6. MEMORIA DE TEXTURAS

La memoria de texturas es utilizada nativamente para leer texturas de imágenes en aplicaciones gráficas. Se puede decir que esta memoria pretende suplir las mismas funcionalidades que la memoria constante (nivel de caché sobre memoria global), pero con restricciones de acceso diferentes. Además, esta caché está organizada en dos niveles y diseñada para optimizar accesos con localidad de datos en dos dimensiones. Con ello, se consigue que los hilos de un mismo half-warp que lean datos próximos en alguna de las dimensiones obtengan un mejor rendimiento. Si los datos resultan no estar en esta caché, será necesario leer de memoria global. Esta memoria es menos restrictiva en cuanto a patrones de acceso, por lo que resulta de gran utilidad cuando nuestro programa no es capaz de adaptarse a las restricciones de la memoria global o constante, siempre que no se requieran escrituras.

4.4. ETAPAS PROGRAMABLES DEL PIPELINE

A pesar de que la tendencia es proporcionar un número más elevado de unidades programables en los procesadores gráficos, las fases que típicamente permiten la programación son las de transformación de vértices y transformación de fragmentos. Tal como muestra la figura 17, estas dos fases permiten programación mediante el procesador de vértices (*vertex shader*) y el procesador de fragmentos (*pixel shader*).

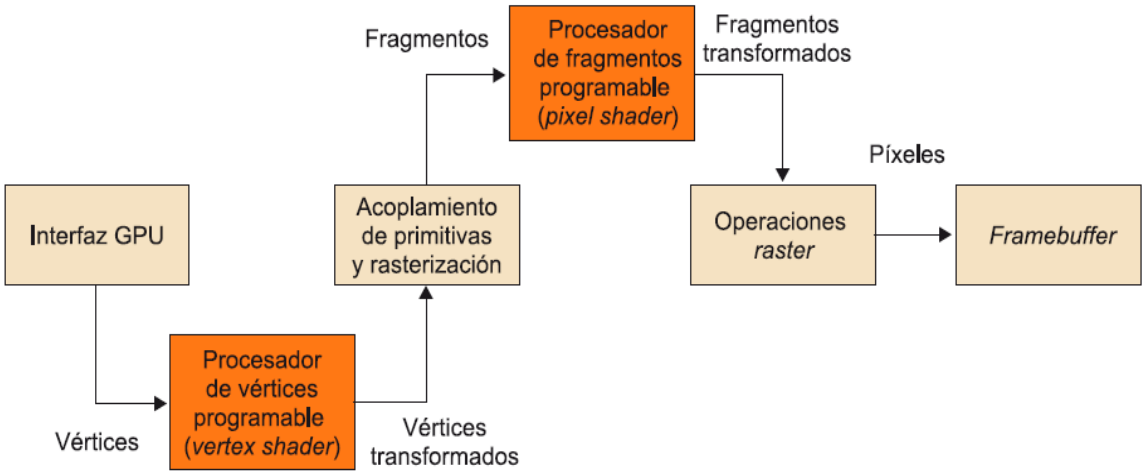


Figura 17. *Pipeline* gráfico programable

El funcionamiento de un procesador de vértices programable es, en esencia, muy similar al de un procesador de fragmentos. El primer paso consiste en la carga de los atributos asociados a los vértices por analizar. Estos se pueden cargar en registros internos del procesador de vértices mismo. Hay tres tipos de registros:

- Registros de atributos de vértices, solo de lectura, con información relativa a cada uno de los vértices.
- Registros temporales, de lectura/escritura, utilizados en cálculos provisionales.

- Registros de salida, donde se almacenan los nuevos atributos de los vértices transformados que, a continuación, pasarán al procesador de fragmentos.

Una vez los atributos se han cargado, el procesador ejecuta de manera secuencial cada una de las instrucciones que componen el programa. Estas instrucciones se encuentran en zonas reservadas de la memoria de vídeo.

Uno de los principales inconvenientes de estos tipos de procesadores programables es la limitación del conjunto de instrucciones que son capaces de ejecutar.

Las operaciones que los procesadores de vértices tienen que poder hacer incluyen básicamente:

- Operaciones matemáticas en coma flotante sobre vectores (ADD, MULT, mínimo, máximo, entre otros).

- Operaciones con hardware para la negación de vectores y swizzling (indexación de valores cuando se cargan de memoria).

- Exponenciales, logarítmicas y trigonométricas.

Los procesadores gráficos modernos soportan también operaciones de control de flujo que permiten la implementación de bucles y construcciones condicionales. Este tipo de procesadores gráficos dispone de procesadores de vértices totalmente programables que funcionan o bien en modalidad SIMD o bien en modalidad MIMD sobre los vértices de entrada.

Los procesadores de fragmentos programables requieren muchas de las operaciones matemáticas que exigen los procesadores de vértices, pero añaden operaciones sobre texturas. Este tipo de operaciones facilita el acceso a imágenes (texturas) mediante el uso de un conjunto de coordenadas para devolver a continuación la muestra leída tras un proceso de filtrado. Este tipo de procesadores

solo funciona en modalidad SIMD sobre los elementos de entrada. Otra característica de estos procesadores es que pueden acceder en modalidad de lectura a otras posiciones de la textura (que corresponderán a un flujo con datos de entrada diferentes). La figura 2 muestra el funcionamiento esquemático de uno de estos procesadores.

4.4.1. INTERFACES DE PROGRAMACIÓN DEL PIPELINE GRÁFICO

Para programar el pipeline de un procesador gráfico de forma eficaz, los programadores necesitan bibliotecas gráficas que ofrezcan las funcionalidades básicas para especificar los objetos y las operaciones necesarias para producir aplicaciones interactivas con gráficos en tres dimensiones.

4.4.2. OpenGL

OpenGL es una de las interfaces más populares. OpenGL está diseñada de manera completamente independiente al hardware para permitir implementaciones en varias plataformas. Esto hace que sea muy portátil, pero no incluye instrucciones para la gestión de ventanas o la gestión de acontecimientos de usuario, entre otros. Las operaciones que se pueden llevar a cabo con OpenGL son principalmente las siguientes (y normalmente también en el orden siguiente):

- Modelar figuras a partir de primitivas básicas, que crean descripciones geométricas de los objetos (puntos, líneas, polígonos, fotografías y mapas de bits).

- Situar los objetos en el espacio tridimensional de la escena y seleccionar la perspectiva desde la que los queremos observar.

- Calcular el color de todos los objetos. El color se puede asignar explícitamente para cada píxel o bien se puede calcular a partir de las condiciones de iluminación o a partir de las texturas.

- Convertir la descripción matemática de los objetos y la información de color asociada a un conjunto de píxeles que se mostrarán por pantalla.

Aparte de estas operaciones básicas, OpenGL también desarrolla otras operaciones más complejas como, por ejemplo, la eliminación de partes de objetos que quedan ocultas tras otros objetos de la escena.

Dada la versatilidad de OpenGL, un programa en OpenGL puede llegar a ser bastante complejo. En términos generales, la estructura básica de un programa en OpenGL consta de las partes siguientes:

- Inicializar ciertos estados que controlan el proceso de renderización.

- Especificar qué objetos se tienen que visualizar mediante su geometría y sus propiedades externas.

El código 1.1 muestra un programa muy sencillo en OpenGL. En concreto, el código del ejemplo genera un cuadro blanco sobre un fondo negro. Como se trabaja en dos dimensiones, no se utiliza ninguna operación para situar la perspectiva del observador en el espacio 3D. La primera función abre una ventana en la pantalla. Esta función no pertenece realmente a OpenGL y, por lo tanto, la implementación depende del gestor de ventanas concreto que se utilice. Las funciones glCleanColor establecen el color actual y glClear borra la pantalla con el color indicado previamente con glCleanColor. La función glColor3f establece qué color se utilizará para dibujar objetos a partir de aquel momento. En el ejemplo, se trata del color blanco (1.0, 1.0, 1.0). A continuación, mediante glOrtho se especifica el sistema de coordenadas 3D que se quiere utilizar para dibujar la escena y cómo se hace el mapeo en pantalla. Después de las funciones glBegin y glEnd, se define la geometría del objeto. En el ejemplo, se definen dos objetos (que aparecerán en la escena) mediante glVertex2f. En este caso, se utilizan coordenadas en dos dimensiones, puesto que la figura que se quiere representar es un plano. Finalmente, la función glFlush asegura que las instrucciones anteriores se ejecuten.

```
#include <GL/gl.h>
#include <GL/glu.h>
void main ()
{
  OpenMainWindow ();
  glClearColor (0.0, 0.0, 0.0, 0.0);
  glClear (GL_COLOR_BUFFER_BIT);
  glColor3f (1.0, 1.0, 1.0);
  glOrtho (-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
  glBegin (GL_POLYGON);
    glVertice2f (-0.5, -0.5);
    glVertice2f (-0.5, 0.5);
    glVertice2f ( 0.5, 0.5);
    glVertice2f ( 0.5, -0.5);
  glEnd ();
  glFlush ();
}
```

Código 1.1. Ejemplo de código en OpenGL

La alternativa directa a OpenGL es Direct3D, que fue presentado en 1995 y finalmente se convirtió en el principal competidor de OpenGL. Direct3D ofrece un conjunto de servicios gráficos 3D en tiempo real que se encarga de toda la renderización basada en software-hardware de todo el *pipeline* gráfico (transformaciones, iluminación y rasterización) y del acceso transparente al dispositivo.

4.4.3. Direct3D

Direct3D es completamente escalable y permite que todo o una parte del pipeline gráfico se pueda acelerar por hardware. Direct3D también expone las capacidades más complejas de las GPU como por ejemplo z-buffering, anti-aliasing, alfa blending, mipmapping, efectos atmosféricos y aplicación de texturas mediante corrección de perspectiva. La integración con otras tecnologías de DirectX permite a Direct3D tener otras características, como las relacionadas con el vídeo, y proporcionar capacidades de gráficos 2D y 3D en aplicaciones multimedia.

Direct3D también tiene un nivel de complejidad bastante elevado. A modo de ejemplo, el código 1.2 muestra cómo podemos definir un cuadrado con Direct3D. Como Direct3D no dispone de ninguna primitiva para cuadrados, como en el caso de OpenGL, se tiene que definir con una secuencia de dos triángulos. Así pues, los tres primeros vértices forman un triángulo y después el resto de vértices añade otro triángulo formado por este vértice y los dos vértices anteriores. Por lo tanto, para dibujar un cuadrado, necesitamos definir cuatro vértices, que corresponden a dos triángulos.

```
Vertice vertices_cuadrado[] ={
  // x, y, z, rhw, color
  { 250.0f, 200.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(255,255,0) },
  { 250.0f, 50.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(255,0,255) },
  { 400.0f, 200.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(0,255,255) },
  { 400.0f, 50.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(255,255,255) }
};
```

Código 1.2: Ejemplo de código en Direct3D para dibujar un cuadrado

4.5. ESTRUCTURA DE UN PROGRAMA EN CUDA

Un programa CUDA está compuesto de una o más fases, las cuales son ejecutadas o en el host o en el dispositivo. Aquellas partes que exhiben poco o nada de paralelismo se implementan en el código a ejecutar sobre el host, no así las que pueden ser resueltas aplicando paralelismo de datos, éstas son implementadas a través de código que se ejecutará en el dispositivo, en este caso la GPU. Si bien en el Programa CUDA existen dos partes bien diferenciadas, será el compilador el responsable de su diferenciación. Para ello, el código 66 MARÍA FABIANA PICCOLI desarrollado para ejecutarse en el host será compilado con el compilador estándar de C (o el del lenguaje secuencial utilizado) y ejecutado en la CPU como un proceso común. El código a ejecutarse en el dispositivo, escrito en C extendido con palabras claves que expresan el paralelismo de datos y las estructuras de datos asociadas, será compilado con el compilador propio de CUDA (nvcc por ejemplo (NVIDIA, 2007)).

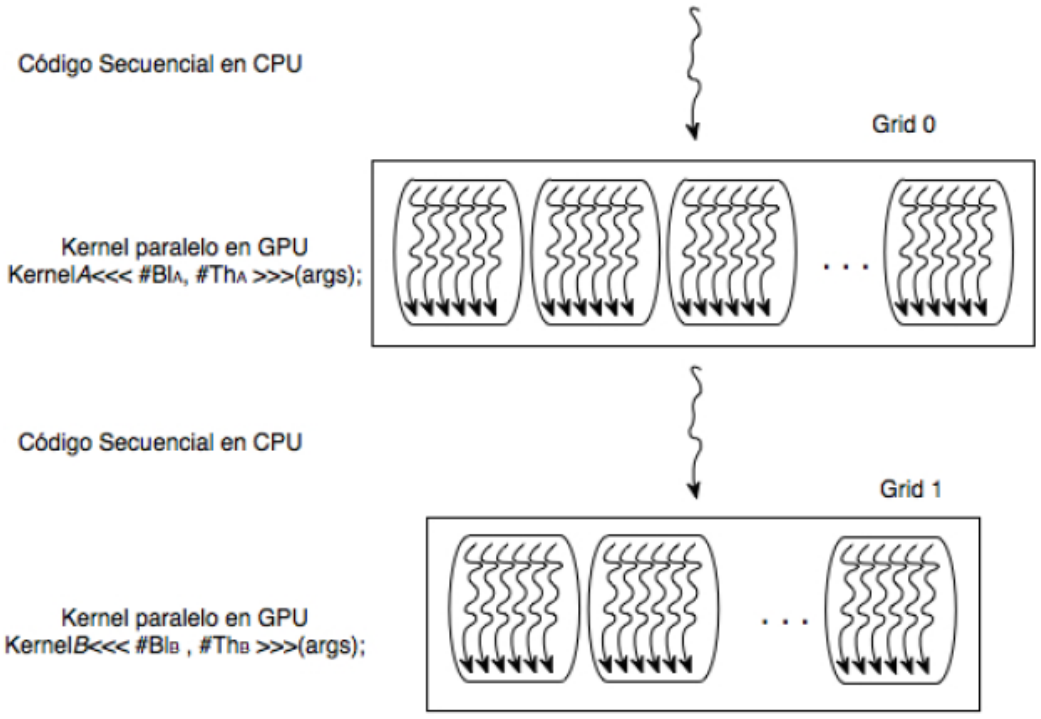


Figura 18: Esquema de la estructura de un programa en CUDA

Un kernel se diferencia de una llamada a una función común en el código secuencial porque además de las especificaciones enunciadas anteriormente, en su invocación se establecen los parámetros necesarios para configurar la ejecución. En la figura 18, se muestra la estructura de un programa CUDA y la forma cómo se realiza la invocación a un kernel, en este caso particular se invocan dos kernel: kernelA y kernelB, indicando para cada caso cuántos bloques (#Bl$_A$, #Bl$_B$) y threads por bloques (#Th$_A$, #Th$_B$) resolverán el problema respectivamente. Dichas especificaciones son realizadas siguiendo la sintaxis:

"función_*kernel*" <<<#Bl, #Th>>>(argumentos);

Donde los argumentos de la función kernel son expresados en (argumentos).

Para poder realizar una explicación más simple, la misma se va a realizar a través de un ejemplo sencillo y apto para ser resuelto en GPU: la suma de dos vectores. Dados dos vectores A y B, su suma algebraica resulta en otro vector C, donde cada

componente de C es la suma de las correspondientes componentes de A y B. En la figura 19 se muestra gráficamente el proceso.

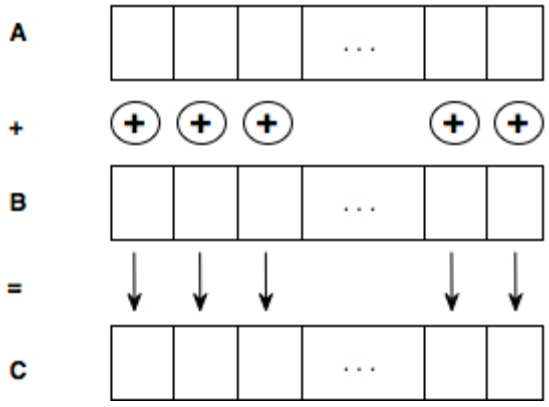


Figura 19: Suma de dos vectores

El siguiente pseudo-código (Figura 20) muestra la solución computacional de la suma de vectores en la CPU.

```
#define N 10
void Suma_vec( int *a, int *b, int *c ) {
  int i = 0; // Los vectores van desde 0..N-1
  for (i=0; i < N; i++)
c[i] = a[i] + b[i];
}
int main( void ) {
  int a[N], b[N], c[N];
inicializa(a);    //Inicializa los vectores de entrada a y b
  inicializa(b);
  Suma_vec( a, b, c );
  mostrar(a,b,c);  // muestra el resultado
  return 0;
}
```

Figura 20: Suma de vectores en CPU

Según lo mostrado en la función *Suma_vec()*, el cálculo de cada una de las componentes del vector C es independiente del resto y, en consecuencia, pueden ser realizados en paralelo. Por ejemplo una solución paralela para una arquitectura con dos núcleos (cores) sería de la forma mostrada la figura 21. Un proceso se encarga de

las componentes pares (figura 21(a)) y el otro de las componentes impares (figura 21(b)).

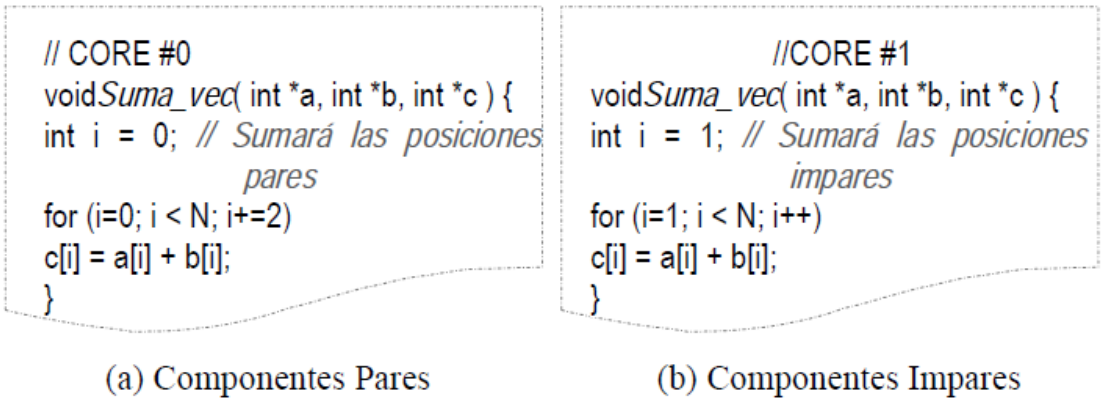


Figura 21: Suma_vec paralela con 2 procesos

En caso de querer resolverlo con más tareas paralelas, se deberá modificar el código de manera tal que permita la creación de procesos para realizar la suma de las componentes, además de contar con el soporte de hardware necesario para la ejecución paralela de los procesos indicados. A continuación se verá cómo resolver el problema en la GPU mediante CUDA C.

Como se dijo antes, un programa CUDA consta de una o más fases, las cuales se ejecutan en la CPU o en la GPU. Aquellas fases que exhiben paralelismo de datos pueden resolverse en paralelo y ejecutarse en la GPU. Al realizarse las llamadas a funciones en la GPU a través de una función kernel, es en ese momento cuando se determina el número de threads con los cuales se resolverá en paralelo el problema. La multiplicación de vectores es una aplicación paralela de datos, por lo cual puede ser resuelta en la GPU. Para ello se debe definir una función kernel donde cada thread es responsable de calcular la suma de un elemento. De esa manera el número de threads dependerá de la dimensión de los vectores a sumar, por ejemplo para sumar vectores de 100 elementos serán necesarios 100 threads, la figura 22 muestra gráficamente cómo se resolverá el problema en la GPU.

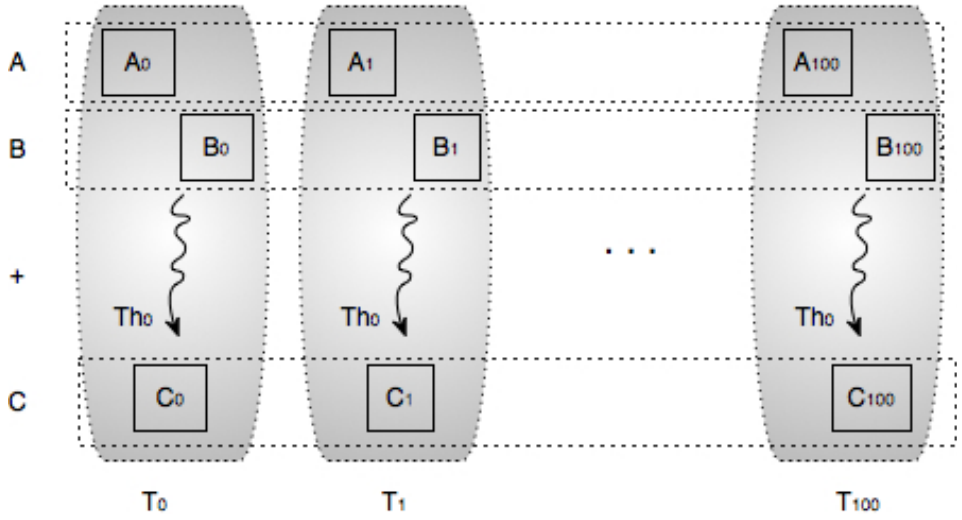


Figura 22: Suma de vectores en GPU

4.5.1. TRANSFERENCIA DE DATOS CPU-GPU

En el sistema CPU-GPU, cada uno de las componentes, host y dispositivo, tienen su propio espacio de memoria, las cuales son independientes. Para resolver un problema en la GPU, el programador necesita transferir los datos de entrada del programa la GPU y, una vez obtenidos los resultados, transferirlos a la CPU. CUDA provee funciones para realizar estas tareas, en las líneas 11 y 12 se lleva a cabo la transferencia de datos desde la CPU a la GPU y en la línea 15 la transferencia es en el sentido inverso. Observe que la función responsable de la transferencia es la misma, la diferencia radica en el último parámetro. Éste indica la constante el sentido de la transferencia a través de una built-in, más adelante se explica con más detalle las transferencias entre las memorias.

Las transferencias de datos entre la CPU y la GPU se dan a nivel de la memoria principal de cada uno.

En la figura 23 se muestra la visión general del modelo de memoria CUDA de la GPU, detallando las posibles transferencias y accesos a los distintos tipos de memoria que anteriormente se han explicado.

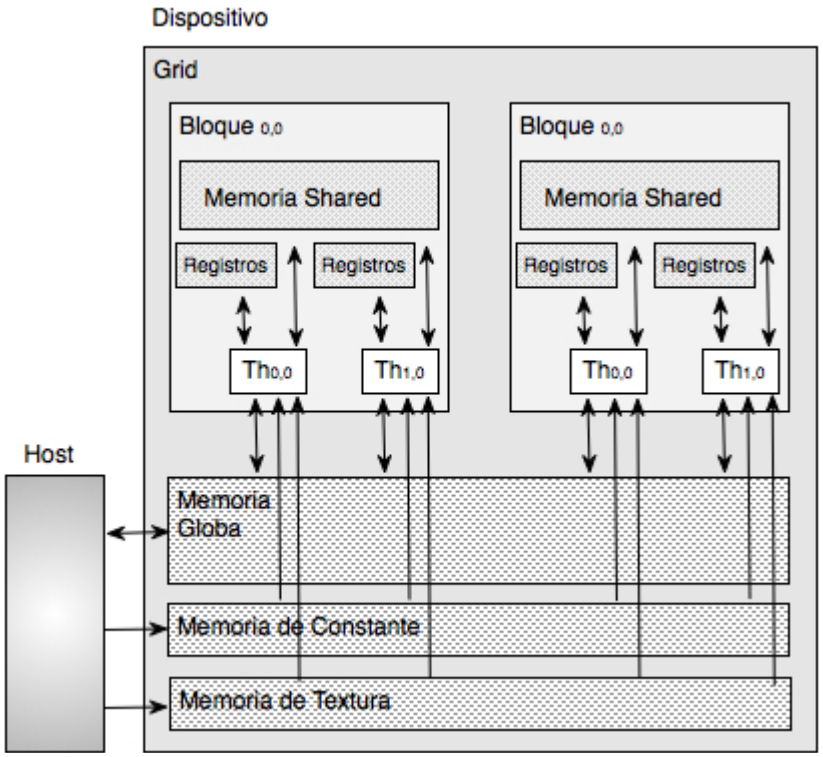


Figura 23: Jerarquía de memoria y accesos

Se puede observar el acceso desde:

- **El host:**
  - A la memoria global del dispositivo, tanto para lectura como para escritura.
  - A la memoria de constante del dispositivo, pero sólo para escritura.
  - A la memoria de textura del dispositivo para sólo escritura.
- **El dispositivo:**
  - A la memoria de registros y a la local, propias de cada thread.
  - A la memoria shared, memoria compartida por todos los threads de un bloque. Cada thread puede leer o escribir en esta memoria.
  - A la memoria de constante y de textura. Estos accesos sólo son de lectura.
  - A la memoria global, a la cual pueden acceder todos los threads tanto para lectura como para escritura.

En esta sección, y para todo este capítulo, sólo se tendrá en cuenta la memoria global.

Para realizar las transferencias a memoria, es necesario gestionar la memoria global en el dispositivo, para ello CUDA provee al programador con funciones para asignar y liberar espacio de memoria en la GPU. En la Figura 3.8 se muestran las funciones de la API para la asignación (líneas 7, 8 y 9) y liberación (líneas 16,17 y 18) de la memoria global. La función cudaMalloc() sólo puede ser llamada en el código del host y permite asignar a una variable una cierta cantidad de espacio, todo es indicado como parámetro en la invocación de la función. Puede observarse una similitud entre cudaMalloc() y la función malloc() de ANSI C. Estas son las características que hacen fácil el aprendizaje de CUDA.

El primer parámetro de cudaMalloc() es la dirección de una variable puntero, la cual contendrá la dirección del objeto asignado en la memoria global después de la llamada. Puede solicitarse espacio para cualquier tipo de objetos. El segundo parámetro especifica el tamaño, en bytes, del objeto a asignar. Para el ejemplo aquí señalado, se puede observar que se reserva lugar para tres vectores, dev_a, dev_b y dev_c. Los dos primeros se corresponden con los vectores de entrada a y b, y en dev_c se calculará el resultado de la suma, el cual será copiado en c. Para mayor claridad, en los ejemplos se antepuso el prefijo “dev_” cuando la variable es del dispositivo.

Para liberar memoria global en la GPU, CUDA provee la función cudaFree(), la cual libera el espacio de memoria apuntado por la variable que recibe como parámetro.

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

Como se mencionó antes, esto se hace a través de la función cudaMemcpy(), la cual tiene cuatro parámetros, ellos son:

- Primer parámetro: Un puntero a la ubicación destino de la copia.

- Segundo parámetro: Puntero a la ubicación desde donde se van a copiar los datos.
- Tercer parámetro: Cantidad de bytes a copiar.
- Cuarto parámetro: Indica el sentido de la transferencia. Esto se hace a través de 4 constantes pre-definidas, ellas son:
  - cudaMemcpyHostToHost: Copia de la memoria principal a la memoria principal.
  - cudaMemcpyHostToDevice: De la memoria principal a la memoria del dispositivo.
  - cudaMemcpyDeviceToHost: Copia desde la memoria del dispositivo a la memoria principal del host.
  - cudaMemcpyDeviceToDevice: De la memoria del dispositivo a la memoria del dispositivo. Este tipo de transferencia es muy rápida.

La función cudaMemcpy() permite realizar copias entre ubicaciones de memoria del host y del dispositivo, no entre varios dispositivos (Ambientes con múltiples GPU). Resumiendo, se muestra la función main() para resolver la suma algebraica de vectores en un sistema CPU-GPU. Es en el host donde se realizan las asignaciones y liberaciones de memoria en el dispositivo, la transferencia de datos entre el host y el dispositivo o a la inversa, y la activación de la ejecución en el dispositivo de la función kernelSuma_vec (para este ejemplo) sobre un determinado número de threads. En la próxima sección se define y analiza la función kernel.

4.6. ESTRATEGIAS Y OPTIMIZACIONES

4.6.1. OPTIMIZACIONES DE PÍXELES

Según Carlos Juega Reimúndez de la Universidad Complutense de Madrid 2010, menciona que si cada hilo carga un pixel en la memoria compartida, entonces todos los hilos que cargan los píxeles del relleno estarán ociosos durante la fase de cómputo. A medida que el radio de la máscara aumenta, el porcentaje de hilos

ociosos aumenta también. Esto desperdicia gran parte del paralelismo, y con la cantidad limitada de memoria compartida, esta pérdida de paralelismo puede ser muy elevada con radios de máscara grandes. Como ejemplo llevado al extremo, vamos a considerar un bloque de imagen de 16x16 y un radio de máscara también de 16. Esta configuración solo permite un bloque de hilos activo por multiprocesador. Asumiendo 4 bytes por pixel, un bloque usará 9216 bytes. Esto es más de la mitad de los 16KB de memoria compartida disponibles por multiprocesador. En este caso, solo 1/9 de los hilos estarán trabajando después de la etapa de carga.

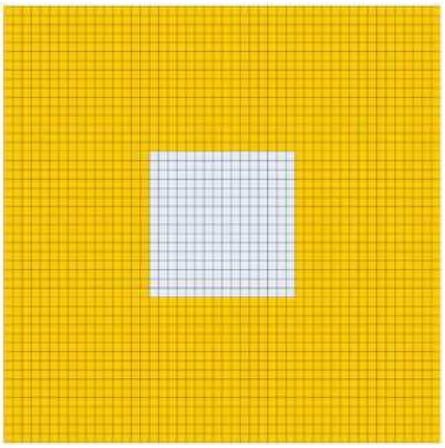


Figura 24: Bloque de Hilos Pixeles

Si el radio de la máscara es grande en comparación al bloque de imagen, habrá muchos hilos ociosos durante la fase de cómputo. Podemos reducir el número de hilos ociosos reduciendo el número de hilos por bloque, y que durante la fase de carga, cada hilo cargue en memoria compartida más de un elemento. Así, podemos usar un hilo por cada elemento a computar. En este caso, para el ejemplo anterior, se divide la matriz en 9 cuadrados de 16x16, de forma que cada hilo lea 9 elementos. Si el relleno no es tan ancho como el bloque de hilos, entonces algunos hilos estarán ociosos en las iteraciones primeras y últimas de la fase de carga.

4.6.2. OPTIMIZAR USO DE MEMORIA

A lo largo de la Sección 4.3 se explicó el funcionamiento del sistema de memoria de la GPU. Cuándo hacer uso de cada nivel de memoria depende

´íntegramente del algoritmo. Por ejemplo, si un algoritmo tiene unos datos de entrada a los que acceden todos los hilos a la vez, tiene sentido que esos datos estén en el espacio de memoria constante. Si por otro lado, los datos a acceder cambian a lo largo del algoritmo, esos datos deben mapearse en el espacio de memoria global y/o compartida. El Cuadro 1 muestra las distintas formas de declarar variables en CUDA, el alcance, el tiempo de vida y su correspondiente mapeo en los distintos niveles de memoria.

Declaración de variable	Memoria	Alcance	Tiempo de vida
variables excepto arrays	registros	hilo	kernel
arrays	global	hilo	kernel
__shared__ int sharedVar;	compartida	bloque	kernel
__device__ int globalVar;	global	grid	aplicación
__constant__ int constVar;	constante	grid	aplicación

Cuadro 1: Variables en CUDA y Memoria

El Cuadro 1 se muestra un hecho de gran importancia: las variables se mapean a registros pero los arrays se mapean a memoria local (que forma parte de la memoria global). Esto, que en las arquitecturas convencionales no tiene ninguna implicación, en la GPU supone que, a alto nivel, el uso de variables escalares es más eficiente que el uso de arrays. Aun así, los datos se suelen alojar en el espacio de memoria global, y por lo tanto, nuestras primeras estrategias de optimización se basarán en acceder a este nivel de memoria.

4.6.3. FLUJO DE INSTRUCCIONES

Las instrucciones de control de flujo tales como saltos tienen también un gran impacto en el rendimiento. Si la condición de salto provoca que hilos del mismo warp tomen caminos distintos, la ejecución deja de ser paralela y deben serializarse ambos caminos del salto.

Primero, los hilos que toman el salto ejecutan una rama y después los hilos restantes ejecutan la otra rama. Por supuesto, esto repercute notablemente en el rendimiento, no sólo por el aumento de instrucciones a ejecutar, sino porque los hilos dejan de ejecutarse en paralelo. Este efecto se ilustra en la Figura 25.

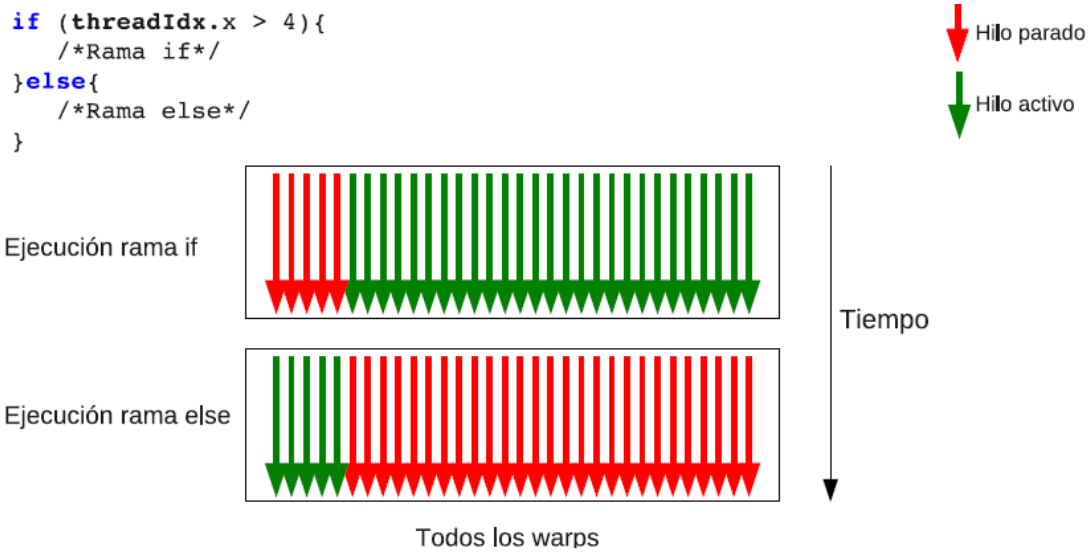


Figura 25: Ejecución en serie de los saltos

Para evitar esto, se debe aumentar la granularidad de los saltos a nivel de warp o un múltiplo. De forma que todos los hilos de un mismo warp sigan el mismo camino y la ejecución continúe siendo paralela.

```
if(threadIdx.x > 4) //provoca que el warp serialice la ejecucion
  ...
else
  ...


if(threadIdx.x/WARP_SIZE > 4) //todos los hilos del mismo warp
  ...                         //siguen el mismo camino y no se
else                          //serializa la ejecucion
  ...
```

Figura 26: Evitar divergencia en los warps

Por último, hablaremos de mejoras a nivel de instrucción. A lo largo de todo el capítulo se ha hablado de lo costosas que son las operaciones en memoria y de

tratar de ocultarlas a base de aumentar el paralelismo. Otro método para ocultarlas es aumentar el paralelismo a nivel de instrucción (ILP). El unrolling de bucles es una técnica con la que es posible aumentar el ILP y reducir el número de instrucciones dinámicas a costa de aumentar el tamaño del código estático y a costa de hacer un mayor uso de registros. Por defecto el compilador trata de desenrollar todos bucles. Sin embargo, en la práctica, sólo es capaz de desenrollar aquellos bucles que en tiempo de compilación tiene un número de iteraciones fijo, lo cual no es habitual. El desenrollado no solo aumenta la relación entre el número de instrucciones de cómputo frente memoria, sino que también elimina las instrucciones de control, evitando la ejecución de instrucciones que no ayudan al progreso del cómputo.

```
for ( int k=0; k<BLOCK SIZE; ++k )
   Pvalue += Ms [ ty ] [ k ] * Ns [ k ] [ tx ];
```

a) El bucle provoca exceso de instrucciones

```
   Pvalue += Ms [ ty ] [ k ] * Ns [ k ] [ tx ] + . . .
                Ms [ ty ] [ k+15]*Ns [ k+15] [ tx ];
```

b) El unrollinge elimina ese exceso

Figura 27: El unrolling mejora la ejecución de instrucciones

Por ejemplo, el bucle de la Figura 27(a) ejecuta 2 instrucciones aritméticas en punto flotante, una instrucción de salto, dos instrucciones aritméticas de direccionamiento y una instrucción para incrementar el contador. Entonces solo 1/3 de las instrucciones ejecutadas son de cómputo del algoritmo. Esta mezcla de instrucciones limita el ancho de banda de procesado de instrucciones, de forma que no se consiga más de 1/3 del potencial.

Sin embargo, en la Figura 27(b) se expresa el mismo cómputo completamente desenrollado. Se consigue eliminar la instrucción de salto y la actualización del

contador. Además como los direccionamientos son constantes, el compilador puede optimizar el código utilizando direccionamiento basado en desplazamientos para eliminar las instrucciones de direccionamiento. Como resultado, se aumenta el rendimiento. Aunque es conveniente realizar el desenrrollado de forma manual para saber exactamente las instrucciones que se ejecutan, no es necesario. Es posible utilizar sentencias de preprocesador antes de los bucles para que la labor de desenrrollado recaiga en el compilador.

CAPÍTULO V

RESULTADOS Y ANALISIS DE RENDIMIENTO

En este último capítulo se presentan los resultados y comparación de rendimiento entre la CPU y GPU, se muestra los beneficios obtenidos de las diferentes funciones implementadas en CUDA. Además muestra los resultados mediante graficas sobre el procesamiento de datos, procesamiento de imágenes, evolución de ancho de bandas así como también la evolución de las arquitecturas Nvidia y AMD.

5.1. COMPARACIÓN DE RENDIMIENTO ENTRE LA CPU Y GPU

Por las propiedades de la GPU, es de esperar que las aplicaciones desarrolladas en ella obtengan buena performance. No siempre es así, las GPU poseen sus limitaciones, las cuales dependen de las características de la arquitectura y, si son desconocidas por el programador, pueden conducir a aplicaciones con bajo rendimiento. En los capítulos anteriores cada vez que se introdujo un concepto, se hizo referencia no sólo a sus características, sino también a las posibles optimizaciones para lograr un buen desempeño.

Desde el punto de vista arquitectural, las primeras generaciones de GPU tenían una cantidad de núcleos bastante reducida, pero rápidamente se incrementó hasta hoy en día, cuando hablamos de dispositivos de tipo many-core con centenares de núcleos en un único chip. Este aumento de la cantidad de núcleos hizo que en el 2003 hubiera un salto importante de la capacidad de cálculo en coma flotante de las GPU respecto a las CPU, tal como muestra la figura 28. Esta figura muestra la evolución del rendimiento en coma flotante (pico teórico) de la tecnología basada en CPU (Intel) y GPU (Nvidia) durante la última década. Se puede apreciar claramente que las GPU van mucho más por delante que las CPU respecto a la mejora de rendimiento, en especial a partir del 2009, cuando la relación era aproximadamente de 10 a 1

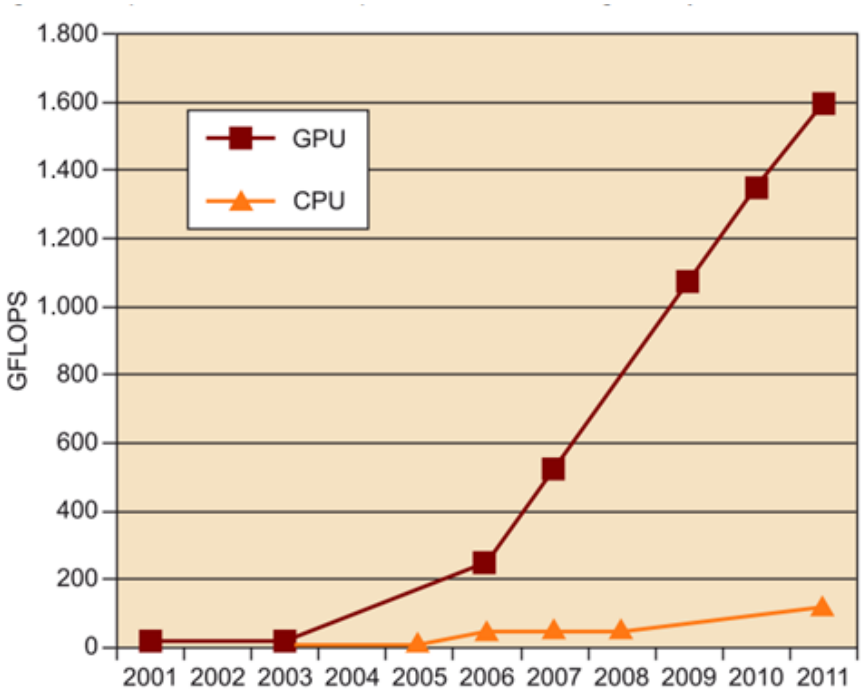


Figura 28: Comparativa de rendimiento entre tecnologías CPU y GPU

Las diferencias grandes entre el rendimiento de CPU y GPU multinúcleo se deben principalmente a una cuestión de filosofía de diseño. Mientras que las GPU están pensadas para explotar el paralelismo a nivel de datos con el paralelismo masivo y una lógica bastante simple, el diseño de una CPU está optimizado para la ejecución eficiente de código secuencial. Las CPU utilizan lógica de control sofisticada que permite un paralelismo a nivel de instrucción y fuera de orden y utilizan memorias caché bastante grandes para reducir el tiempo de acceso a los datos en memoria. También hay otras cuestiones, como el consumo eléctrico o el ancho de banda de acceso a la memoria. Las GPU actuales tienen anchos de banda a memoria en torno a diez veces superiores a los de las CPU, entre otras cosas porque las CPU deben satisfacer requisitos heredados de los sistemas operativos, de las aplicaciones o dispositivos de entrada/salida.

Esto también ha habido una evolución muy rápida desde el punto de vista de la programación de las GPU que ha hecho cambiar el propósito de estos dispositivos. Las GPU de principios de la década del 2000 utilizaban unidades aritméticas

programables (shaders) para devolver el color de cada píxel de la pantalla. Como las operaciones aritméticas que se aplicaban a los colores de entrada y texturas las podía controlar completamente el programador, los investigadores observaron que los colores de entrada podían ser cualquier tipo de dato. Así pues, si los datos de entrada eran datos numéricos que tenían algún significado más allá de un color, los programadores podían ejecutar cualquiera de los cálculos que necesitaran sobre esos datos mediante los shaders.

A pesar de las limitaciones que tenían los programadores para desarrollar aplicaciones sobre GPU (por ejemplo, escribir resultados en cualquier dirección de memoria), el alto rendimiento con operaciones aritméticas hizo que se dedicaran muchos esfuerzos a desarrollar interfaces y entornos de programación de aplicaciones de propósito general para GPU. Algunas de estas interfaces de programación han tenido mucha aceptación en varios sectores, a pesar de que su uso todavía requiere de una cierta especialización.

5.2. COMPARACIÓN ENTRE EL PIPELINE SECUENCIAL (IZQUIERDA) Y EL CÍCLICO DE LA ARQUITECTURA UNIFICADA (DERECHA).

La aparición de la arquitectura G80 representó una mejora de la capacidad de procesamiento gráfico y un incremento de las prestaciones respecto a la generación de GPU. Gracias a la arquitectura unificada, el número de etapas del pipeline se reduce de manera significativa y pasa de un modelo secuencial a un modelo cíclico, tal como muestra la figura 29.

El pipeline clásico utiliza diferentes tipos de shaders por medio de los cuales los datos se procesan secuencialmente. En cambio, en la arquitectura unificada solo hay una única unidad de shaders no especializados que procesan los datos de entrada (en forma de vértices) por pasos. La salida de un paso retroalimenta los shaders que pueden ejecutar un conjunto diferente de instrucciones, de este modo se emula el

pipeline clásico, hasta que los datos han pasado por todas las etapas del pipeline y se encaminan hacia la salida de la unidad de procesamiento.

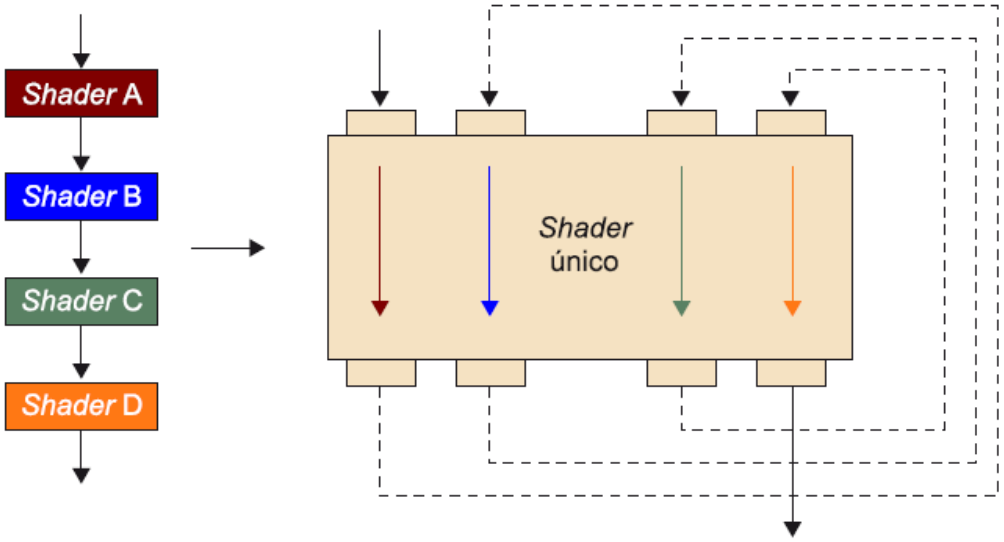


Figura 29: Comparación entre el pipeline secuencial (izquierda) y el cíclico de la arquitectura unificada (derecha)

El funcionamiento básico en la arquitectura G80 para ejecutar un programa consiste en dos etapas diferenciadas. En la primera etapa, los datos de entrada se procesan mediante hardware especializado. Este se encarga de distribuir los datos de tal manera que se puedan utilizar el máximo número de unidades funcionales para obtener la máxima capacidad de cálculo durante la ejecución del programa. En la segunda etapa, un controlador global de flujos se encarga de controlar la ejecución de los flujos que ejecutan los cálculos de manera coordinada. También determina en cada momento qué flujos y de qué tipo (de vértices, de fragmentos o de geometría) serán enviados a cada unidad de procesamiento que compone la GPU. Las unidades de procesamiento tienen un planificador de flujos que se encarga de decidir la gestión interna de los flujos y de los datos.

5.3. COMPARACION DE MODELOS DE NVIDIA Y AMD

Implementaciones posteriores a la serie G80 ofrecen todavía más prestaciones, como un número más elevado de SP, más memoria o ancho de banda.

La tabla 1 muestra las principales características de algunas de las implementaciones dentro de esta familia de Nvidia. Las últimas generaciones de Nvidia implementan la arquitectura Fermi, que proporciona soluciones cada vez más masivas a nivel de paralelismo, tanto desde el punto de vista del número de SP como de flujos que se pueden ejecutar en paralelo.

Modelo	Arqui-tectura	Reloj (MHz)	Núcleos		Memoria				Rendi-miento - pico teórico (GFLOP)
			Número de SP	Reloj (MHz)	Tipo	Tamaño (MB)	Reloj (MHz)	Ancho de banda (GB/s)	
C870	G80	600	128	1.350	GDDR3	1.536	1.600	76	518
C1060	GT200	602	240	1.300	GDDR3	4.096	1.600	102	933
C2070	GF100	575	448	1.150	GDDR5	6.144	3.000	144	1.288
M2090	GF110	650	512	1.300	GDDR5	6.144	3.700	177	1.664

Tabla 1: Comparativa de varios modelos de la familia Tesla de Nvidia

La tabla 2 muestra las características principales de las arquitecturas AMD orientadas a GPGPU. Notad que el número de SP es mucho más elevado que en las Nvidia, pero también existen diferencias importantes respecto a la frecuencia de reloj y ancho de banda en la memoria.

Modelo	Arquitectura (familia)	Núcleos		Memoria				Rendimiento - pico teóri-co (GFLOP)
		Número de SP	Reloj (MHz)	Tipo	Tamaño (MB)	Reloj (MHz)	Ancho de banda (GB/s)	
R580	X1000	48	600	GDDR3	1.024	650	83	375
RV670	R600	320	800	GDDR3	2.048	800	51	512
RV770	R700	800	750	GDDR5	2.048	850	108	1.200
Cypress (RV870)	Evergreen	1.600	825	GDDR5	4.096	1.150	147	2.640

Tabla 2: Comparativa de varios modelos de GPU AMD

5.4. OPERACIONES DE PUNTO FLOTANTE POR SEGUNDO GPU vs CPU

Sometidos a la demanda insaciable de mercado en cómputo en tiempo real, alta definición de gráficos en 3D, el procesador de gráficos programable o GPU se ha

convertido en un procesador de múltiples núcleos altamente paralelos, multiproceso, con enorme poder de cómputo (figura 30) y ancho de banda de memoria muy alto.

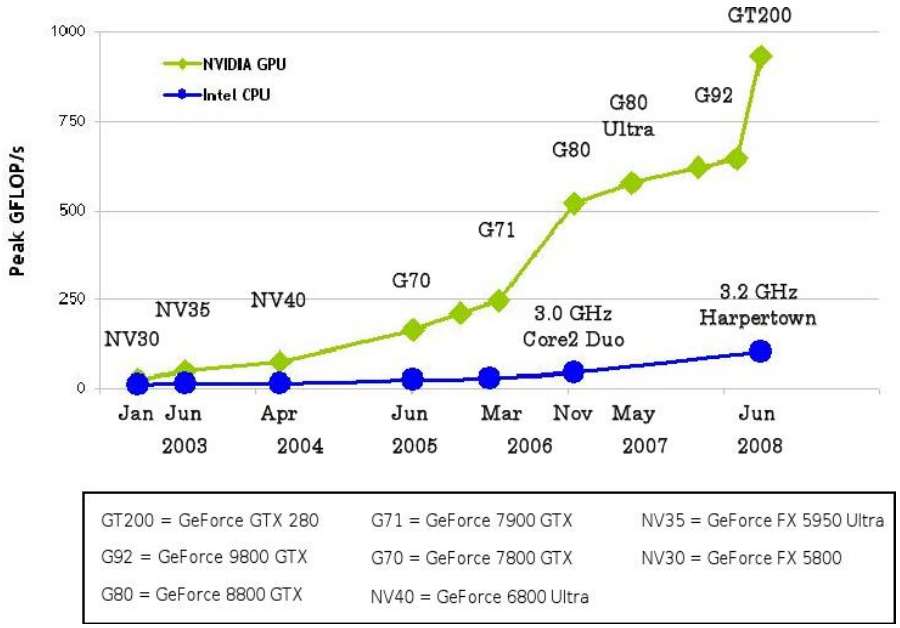


Figura 30: Operaciones de punto flotante por segundo (CPU y GPU)

La razón de la discrepancia en la capacidad de operaciones en punto flotante entre la CPU y la GPU es que la GPU está especializada en cómputo intensivo, computación altamente paralela y por lo tanto diseñada de manera que más transistores se dedican al procesamiento de datos (figura 31) en lugar de almacenamiento en cache de datos y control de flujo.

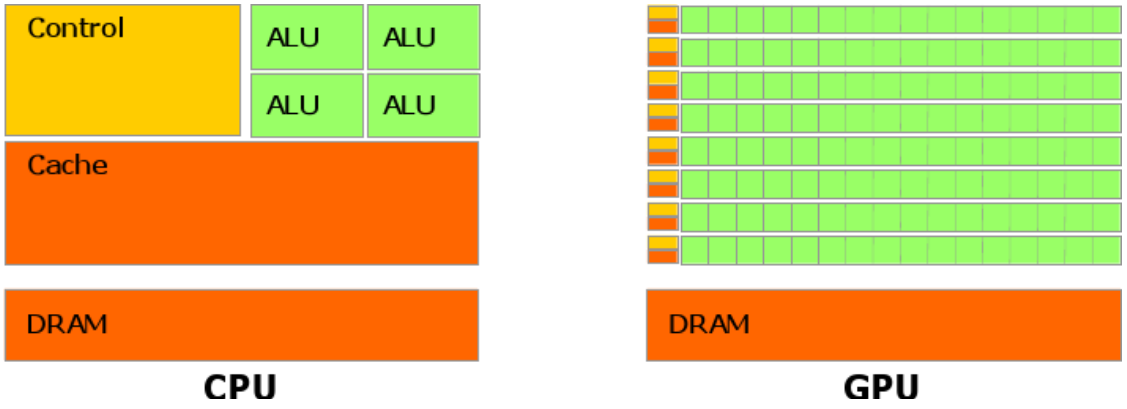


Figura 31: La GPU con más transistores para el procesamiento de datos

5.4.1. PROCESAMIENTO DE IMAGENES

En la siguiente figura, se muestra una gráfica de Comparaciones de transferencias de imágenes en una GPU y en una CPU, considerando el tamaño de las imágenes, en este caso el tamaño esta medido por dimensiones de pixeles.

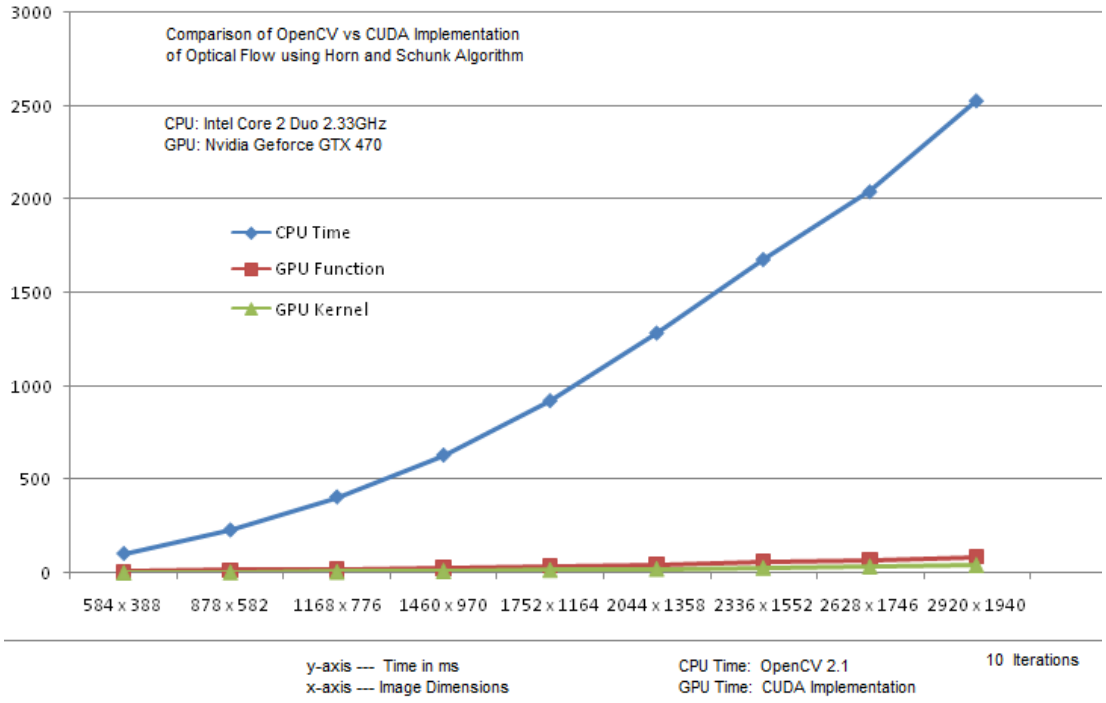


Figura 32: Procesamiento de imágenes en una GPU vs CPU

5.5. APLICACIONES DE CUDA

En la actualidad, decenas de miles de desarrolladores, científicos, estudiantes, creadores de juegos e investigadores realizan aplicaciones que aprovechan la computación de la GPU en áreas tan diversas como juegos basados en la física, análisis de riesgo de activos, análisis de datos sísmicos y pronósticos climáticos. Algunas de ellas son las siguientes:

5.5.1. Aplicaciones de CUDA (Energía):

SeismicCity: Utiliza CUDA para mejorar la posibilidad de descubrir petróleo. El costo de la perforación en la exploración profunda de pozos de petróleo puede llegar

a cientos de millones de dólares. En muchos casos, existe apenas una posibilidad de perforar un pozo con éxito. La tecnología de imágenes en profundidad basada en CUDA (figura 33) de SeismicCity interpreta datos sísmicos que llevan a la selección de nuevas ubicaciones para perforación con mucho más rapidez de lo que podrían hacerlo los sistemas anteriores. Para mejorar la calidad y eficiencia de sus imágenes SeismicCity se inclinó por CUDA y las GPU NVIDIA Tesla de la serie 8. Con ello se logró un aumento de hasta 14 veces en el rendimiento con relación a la configuración anterior basada en la CPU.

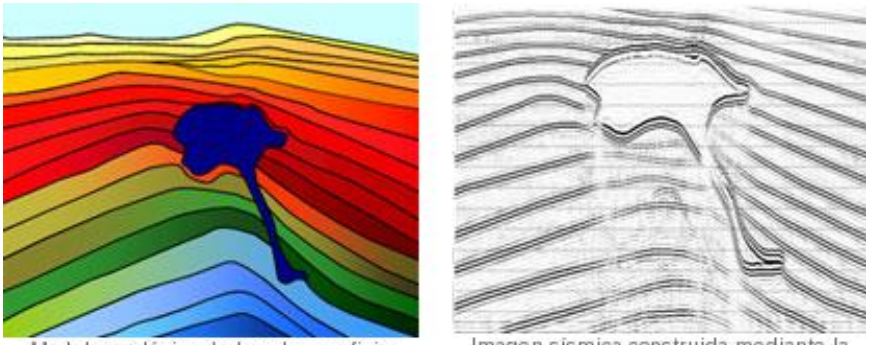
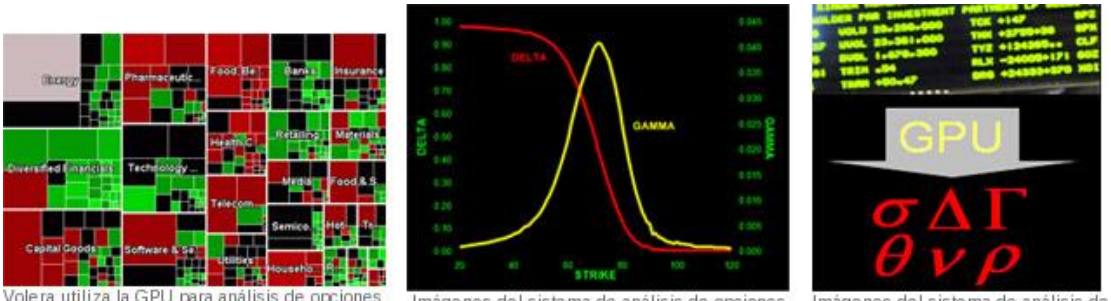


Figura 33: Aplicaciones de CUDA (Energía)

5.5.2. Aplicaciones de CUDA (Finanzas)

Para Hanweck Associates una firma de servicios financieros especializada en la administración de riesgo e inversiones, es esencial ofrecerles a los clientes una forma de recalcular las opciones en tiempo real. Hanweck lo logra a través de su línea Volera (figura 34) de análisis de opciones de alto rendimiento. Usando apenas 12 GPU aptas para CUDA, Volera analiza el mercado completo de opciones de capital de los EE UU. En tiempo real, una tarea que antes necesitaba más de 60 servidores convencionales.



Volera utiliza la GPU para análisis de opciones en tiempo real

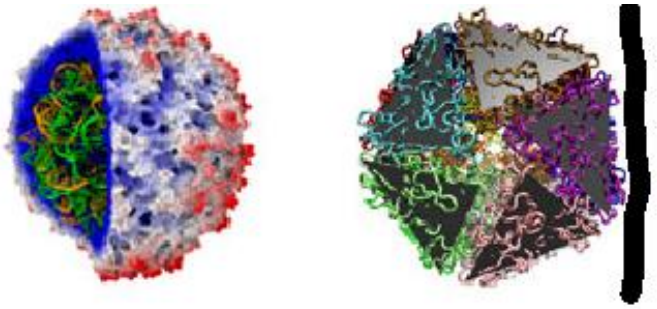
Imágenes del sistema de análisis de opciones de Volera

Imágenes del sistema de análisis de opciones de Volera

Figura 34: Aplicaciones de CUDA (Finanzas)

5.5.3. Aplicaciones de CUDA (Investigación)

Los virus, causa de muchas enfermedades, son los organismos naturales más pequeños que se conocen. Debido a su simplicidad y al tamaño pequeño, los biólogos computacionales eligieron un virus para su primer intento de simular una forma de vida completa usando una computadora. Se trata del virus satélite de mosaico del tabaco, uno de los más pequeños. Los investigadores simularon el virus en una gota de agua salada usando un programa llamado NAMD (Nanoscale Molecular Dynamics o dinámica molecular en nano escala) de la Universidad de Illinois en Urbana-Champaign. Las aplicaciones NAMD se han acelerado con CUDA (figura 35), logrando aumentos impresionantes de hasta 330 veces en la velocidad, en comparación con una CPU de núcleo único al ejecutarse en un clúster acelerado por la GPU en el Centro Nacional de Aplicaciones de Supercomputación (NCSA).



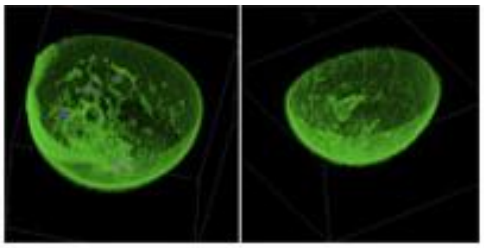
Capa protectora de la proteína que envuelve el genoma viral

Estructura icosahédrica de la capa de la proteína

Figura 35: Aplicaciones de CUDA (Investigación)

5.5.4. Aplicaciones de CUDA (Medicina)

La capacidad de producir rápidamente imágenes bastante detalladas en un plazo corto tiene gran relevancia en el campo de las ecografías para detección del cáncer de mama. TechniScan, un desarrollador de sistemas de imágenes automatizadas por ultrasonido, cambió su algoritmo patentado de un sistema tradicional en la CPU a CUDA y a las GPU NVIDIA Tesla. El sistema basado en CUDA (figura 36) es capaz de procesar el algoritmo de TechniScan dos veces más rápido.



Renderizado volumétrico del sistema de ultrasonido para la mama completa (WBU) de TechniScan.

Figura 36: Aplicaciones de CUDA (Medicina)

5.5.5. Aplicaciones de CUDA (Vídeo y Fotografía)

A medida en que crece la popularidad de los dispositivos de medios digitales, los usuarios experimentan mayor frustración por la demora en la tarea de colocar vídeo en sus dispositivos. Por ejemplo, convertir una película de dos horas de duración puede tardar seis horas o más cuando se usa la CPU de la computadora. Badaboom de Elemental es un programa de transcodificación de vídeo que convierte los archivos de vídeo estándar en formatos que se ejecutan en el Ipod y en otros dispositivos portátiles. Al aprovechar CUDA en las GPU NVIDIA, Badaboom puede acelerar el proceso de conversión para que resulte hasta 18 veces más rápido que los métodos tradicionales. La conversión de una película de dos horas de duración tarda cerca de 20 minutos (figura 37) en vez de varias horas.

Figura 37: Aplicaciones de CUDA (Vídeo y Fotografía)

Además, el mundo académico ha reconocido el increíble potencial de esta arquitectura de computación. La computación de la GPU basada en CUDA ahora forma parte del plan de estudios de más de 20 universidades, entre las que se incluyen el MIT, Harvard, Cambridge, Oxford, los Institutos de Tecnología de la India, la Universidad Nacional de Taiwan y la Academia China de Ciencias.

5.6. PRESTACIONES CPU-GPU

El reloj de los SPs y SFUs en la GeForce 8800 Ultra es de 1.5 GHz, para unas prestaciones pico de 576 GFLOPS. Por otra parte, la GeForce GTX tiene un reloj de 1.35 GHz y unas prestaciones pico de 518 GFLOPS.

A continuación se comparan las prestaciones de la GPU GeForce 8800 frente a una CPU multinucleo con tres aplicaciones diferentes de algebra lineal densa, transformada rápida de Fourier y ordenación. Los programas y bibliotecas de la GPU se han implementado con CUDA y C. Los programas de la CPU utilizan la biblioteca Intel MKL 10.0 multihilo de precisión simple para obtener una implementación eficiente con instrucciones SSE y varios núcleos.

David PATTERSON (2011), presentan resultados de prestaciones en GPU y CPU para varias operaciones sobre matrices densa en precisión simple: multiplicación de matriz-matriz (la rutina SGEMM) y factorizaciones LU, QR y Cholesky. En la Figura 38 se comparan los GFLOPS obtenidos en la multiplicación SGEMM de matrices densa de una GPU GeForce 8800 GTX y una CPU de cuatro

núcleos. Asimismo, la Figura 39, muestra la comparación de ejecución de varios algoritmos de factorización.

Como la mayor carga computacional de la factorización de matrices corresponde a la multiplicación de matriz-matriz de SGEMM y rutinas similares en BLAS3, sus presiones fijan un límite superior en el ritmo de factorización.

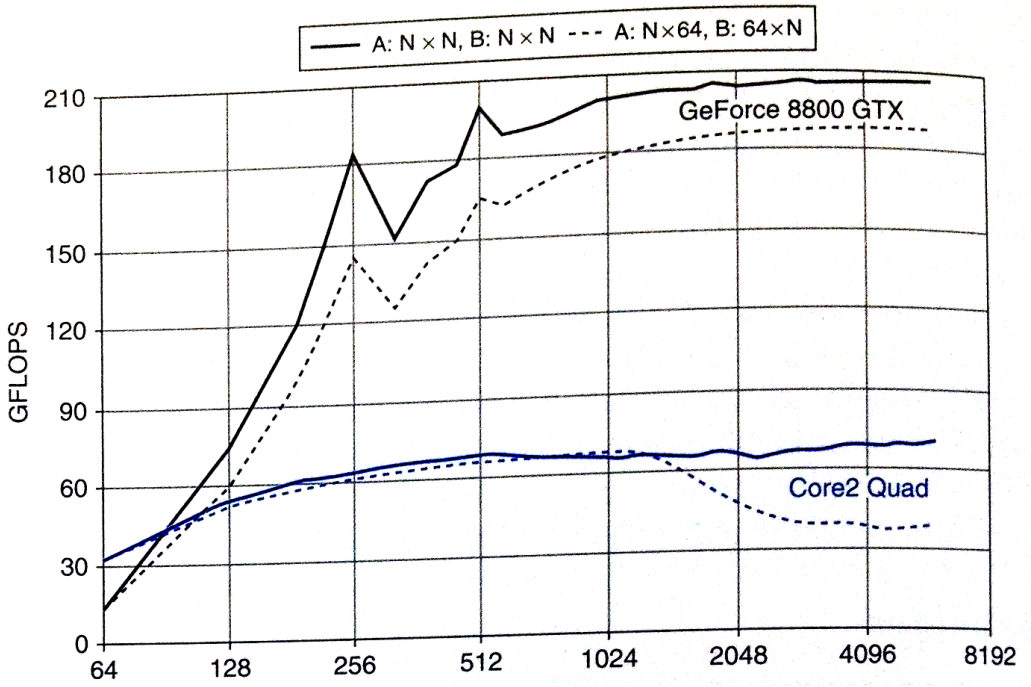


Figura 38: Prestaciones de la Multiplicación SGEMM de matrices densas de una GPU GeForce 8800 GTX y una CPU

La grafica muestra los GFLOPS obtenidos en la multiplicación de matrices cuadradas NxN (línea solida) y matrices Nx64 y 64xN (líneas punteadas). Extraídas de la figura A.7.3 de David PATTERSON (2011), Las gráficas en color negro corresponde a una GeForce 8800 GTX a 1.35 GHz ejecutado el código SGEMM de Volkov (actualmente en CUBLAS 2.0 de NVIDIA) con matrices en la memoria de la GPU. Las gracias en azul corresponden a la ejecución de un Intel Core2 Quad Q6600 de cuatro núcleos a 2.4 GHz, con Linux de 64 bits y la biblioteca Intel MKL con matrices en la memoria de la CPU.

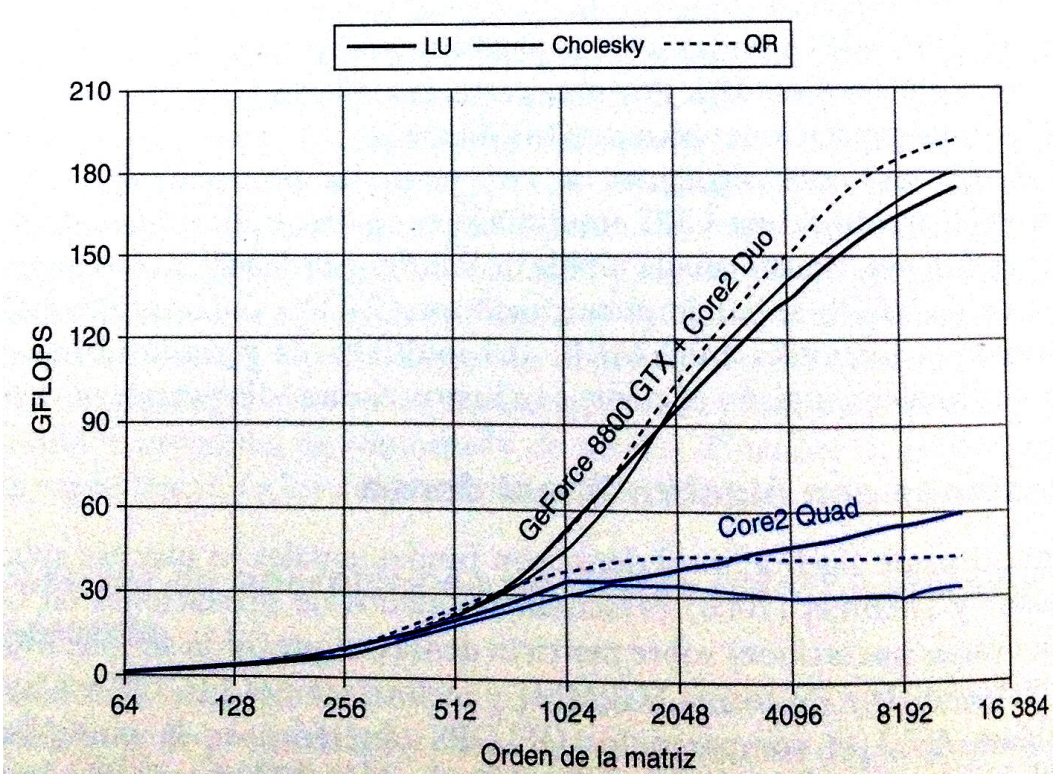


Figura 39: Prestaciones de la factorización de matrices densas.

Las gráficas muestran los GFLOPS alcanzados en factorización de matrices utilizando la GPU y utilizando solo la CPU. Extraída de la figura A.7.4 de David PATTERSON (2011). Las gráficas en color negro corresponden a un sistema con una GeForce 8800 GTX de NVIDIA a 1.35 GHz, CUDA 1.1, Windows XP y un procesador Intel Core 2 Duo E6700 a 2.67 GHz con Windows XP, incluyendo en el tiempo de transferencia de datos entre CPU y GPU. Las gráficas en azul corresponden a la ejecución en un Intel Core2 Quad Q6600 de cuatro núcleos a 2.4.GHz, con Linux 64 Bits y la biblioteca Intel MKL.

5.6.1. PRESTACIONES DE CPU/GPU EN CÁLCULO Y ACCESO A MEMORIA

Procesador	CPU (AMD)	GPU (Nvidia)
Modelo arquitectural	Opteron X2 2218	G80
Frecuencia de reloj	2.6 GHz	600 MHz / 1.35 GHz
Número de núcleos	2 cores	128 stream processsors
Potencia de cálculo	2 cores x 4.4 GFLOPS = 8.8 GFLOPS	madd(2 FLOPS) x128 SP x 1.35 GHz = 345.6 GFLOPS
En el total de 18 nodos x 2 zócalos	316.8 GFLOPS	¡¡ 12.4 TFLOPS !!

Memoria	CPU (AMD)	GPU (Nvidia)
Capacidad y tipo	8 Gbytes de DDR2	1.5 Gbytes de GDDR3
Frecuencia de reloj	2x 333 MHz	2x 800 MHz
Anchura del bus	128 bits (doble canal)	384 bits
Ancho de banda	10.8 Gbytes/sg.	76.8 Gbytes/sg.

Figura 40: cálculo y acceso a memoria

5.7. Evolución GPU: GB/s.

En la figura se puede apreciar cómo ha evolucionado el ancho de banda en las GPU vs CPU.

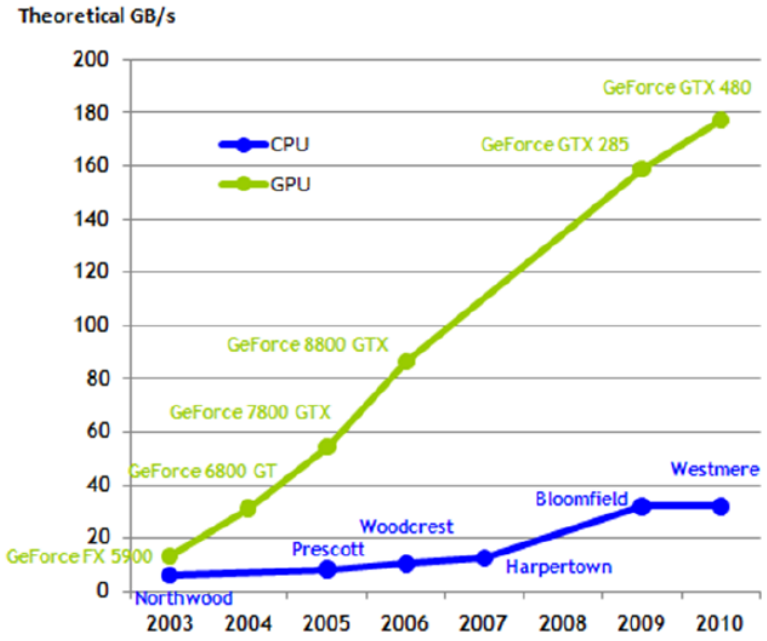


Figura 41: Ancho de banda en las GPU vs CPU

CONCLUSIONES

La complejidad de un solo Core ha llegado a tal punto que no es posible sacar más rendimiento con variaciones arquitectónicas. Por tanto, ya no puede esperarse que un mismo binario incremente sus prestaciones automáticamente con cada familia de procesadores. Los fabricantes, al no saber qué hacer con el área del que disponen, han apostado claramente hacia la replicación de cores. En ese contexto, las GPUs han emergido como plataformas de procesamiento masivamente paralelas. Si bien no son de propósito general, hay una gran variedad de aplicaciones que pueden beneficiarse de sus características.

Este tipo de herramientas deben ser capaces de llegar a soluciones óptimas. Para ello, deben saber cómo explotar los recursos disponibles en la GPU, entre ellos cabe destacar el sistema heterogéneo de memoria y su alta latencia, el diseño masivamente paralelo del procesador, en el que miles de hilos pueden colaborar con un objetivo común, optimizaciones sobre el flujo de instrucciones y el paralelismo a nivel de instrucción.

Desde sus inicios, la GPU fue diseñada como un procesador paralelo. El pipeline gráfico implementado por las primeras GPU provee paralelismo a nivel de etapas. Desde sus inicios se diferenció de las CPU, sus filosofías de desarrollo son distintas, los avances de las CPU obedecen a ofrecer mejores prestaciones y reducir el tiempo de las aplicaciones secuenciales existentes. No ocurre lo mismo con las GPU donde los avances se deben a la necesidad de optimizar las aplicaciones paralelas.

En ambos casos, el número de cores seguirá aumentando en la medida que sea posible. Por su parte, las GPUs continuarán disfrutando de la evolución arquitectónica vigorosa. Es de esperar que la brecha entre ambas arquitecturas se siga manteniendo en un tiempo en el futuro. Respecto al rendimiento de las aplicaciones desarrolladas en ambas sistemas, en el caso de las CPU es necesario instaurar a la computación paralela como la alternativa. Respecto a la GPU, si bien su utilización en aplicaciones

de propósito general es muy reciente, los rendimientos alcanzados son muy buenos. Esto unido a los constantes avances en las nuevas arquitecturas y el constante desarrollo de herramientas que facilitan su programación, permiten enunciar que se constituirán en un recurso económico y válido para el desarrollo de aplicaciones en general.

Finalmente ante el desarrollo de una arquitectura más general de la GPU, existen diferentes herramientas, las cuales permiten una programación genérica igual que ocurre con la CPU. Una de las herramientas más populares para la programación de propósito general es CUDA, definido por Nvidia para sus GPU a partir de la serie G80.

RECOMENDACIONES

Durante las últimas décadas, uno de los métodos más relevantes para mejorar el rendimiento de los computadores ha sido el aumento de la velocidad del reloj del procesador. Sin embargo, los fabricantes se vieron obligados a buscar alternativas a este método debido a varios factores como, por ejemplo:

• Limitaciones fundamentales en la fabricación de circuitos integrados como, por ejemplo, el límite físico de integración de transistores.

• Restricciones de energía y calor debidas, por ejemplo, a los límites de la tecnología CMOS y a la elevada densidad de potencia eléctrica.

• Limitaciones en el nivel de paralelismo a nivel de instrucción (instruction level parallelism). Esto implica que, haciendo el pipeline cada vez más grande, se puede acabar obteniendo peor rendimiento.

BIBLIOGRÁFICA

[1] *Arquitecturas basadas en computación gráfica (GPU).* Francesc Guim e Ivan Rodero (2011), de la Universidad Abierta de Cataluña.

[2] *Estructura y Diseño de Computadores*. David A. Patterson y John L. Hennessy (2011), Editorial Reverte.

[3] *Computación de alto desempeño en GPU*. María Fabiana Piccoli (2011), de la Universidad Nacional de la Plata.

[4] *Estudio de rendimiento en GPU*. Carlos Juega Reimúndez (2010), de la Universidad Complutense de Madrid.

[5] *Implementación CPU-GPU y comparativa de las bibliotecas BLAS-CUBLAS, LAPACK-CULA.* Misael Ángeles Arce, Georgina Flores Becerra, and Antonio M. Vidal.