



UNIVERSIDAD
DE MÁLAGA

Departamento de Arquitectura de Computadores

Programa de Doctorado
Ingeniería Mecatrónica

TESIS DOCTORAL

Planificación Concurrente de Comandos en GPU

Bernabé López Albelda

Abril de 2023

Dirigida por:
Nicolás Guil Mata
José M^a González Linares


UNIVERSIDAD
DE MÁLAGA





UNIVERSIDAD
DE MÁLAGA

AUTOR: Bernabé López Albelda

 <https://orcid.org/0000-0001-6153-7651>

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga



Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional:

<https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode>

Cualquier parte de esta obra se puede reproducir sin autorización pero con el reconocimiento y atribución de los autores.

No se puede hacer uso comercial de la obra y no se puede alterar, transformar o hacer obras derivadas.

Esta Tesis Doctoral está depositada en el Repositorio Institucional de la Universidad de Málaga (RIUMA): riuma.uma.es





DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D./Dña BERNABÉ LÓPEZ ALBELDA

Estudiante del programa de doctorado INGENIERÍA MECATRÓNICA de la Universidad de Málaga, autor/a de la tesis, presentada para la obtención del título de doctor por la Universidad de Málaga, titulada: PLANIFICACIÓN CONCURRENTE DE COMANDOS EN GPU

Realizada bajo la tutorización de DR. D. NICOLÁS GUIL MATA y dirección de DR. D. NICOLÁS GUIL MATA Y DR. D. JOSÉ MARÍA GONZÁLEZ LINARES (si tuviera varios directores deberá hacer constar el nombre de todos)

DECLARO QUE:

La tesis presentada es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, conforme al ordenamiento jurídico vigente (Real Decreto Legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), modificado por la Ley 2/2019, de 1 de marzo.

Igualmente asumo, ante a la Universidad de Málaga y ante cualquier otra instancia, la responsabilidad que pudiera derivarse en caso de plagio de contenidos en la tesis presentada, conforme al ordenamiento jurídico vigente.

En Málaga, a 17 de ABRIL de 2023

Fdo.: Doctorando/a	Fdo.: Tutor/a
Fdo.: Director/es de tesis	





UNIVERSIDAD
DE MÁLAGA



Escuela de Doctorado



EFQM AENOR



Edificio Pabellón de Gobierno. Campus El Ejido.
29071
Tel.: 952 13 10 28 / 952 13 14 61 / 952 13 71 10

E-mail:

doctorado@uma.es

-

Dr. D. Nicolás Guil Mata.
Catedrático del Departamento de Ar-
quitectura de Computadores de la Uni-
versidad de Málaga.

Dr. D. José M^a González Linares.
Profesor Titular del Departamento de
Arquitectura de Computadores de la
Universidad de Málaga.

CERTIFICAN:

Que la memoria titulada “Planificación Concurrente de Comandos en GPU”,
ha sido realizada por D. Bernabé López Albelda bajo nuestra dirección en el
Departamento de Arquitectura de Computadores de la Universidad de Málaga
y constituye la Tesis que presenta para optar al grado de Doctor en Ingeniería
Mecatrónica.

Málaga, Abril de 2023

Dr. D. Nicolás Guil Mata.
Codirector de la tesis.

Dr. D. José M^a González Linares.
Codirector de la tesis.



UNIVERSIDAD
DE MÁLAGA



UNIVERSIDAD
DE MÁLAGA

Autorización para la lectura de la Tesis e Informe de la utilización de las publicaciones que la avalan

Los abajo firmantes declaran, bajo su responsabilidad, que autorizan la lectura de la tesis del doctorando D. Bernabé López Albelda, con DNI _____ titulada Planificación Concurrente de Comandos en GPU y que ninguna de las publicaciones que avalan dicha tesis ha sido utilizada en tesis anteriores.

Málaga, Abril de 2023

Dr. Nicolás Guil Mata
Tutor y codirector de la tesis

José María González Linares
Codirector de la tesis

UNIVERSIDAD
DE MÁLAGA





UNIVERSIDAD
DE MÁLAGA

A Cristina, mis padres y hermanos.



UNIVERSIDAD
DE MÁLAGA

Agradecimientos

Estos años en los que he estado realizando la tesis, he convivido con muchas personas a las que quiero agradecer su compañía.

En primer lugar, agradecer a mis directores Nicolás y José María por todo su apoyo y trabajo durante este tiempo. Los conocí durante mi paso por el Máster en Ingeniería Informática en la Universidad de Málaga y, para mí, son un ejemplo de dedicación, capacidad de trabajo y entrega que muestra su gran desempeño como docentes e investigadores. Muchas gracias.

Extender mis agradecimientos a mis compañeros de laboratorio con los que he podido establecer una gran amistad: Fran, Rubén, Paula, José Carlos, Iván, Andrés, Herruzo, Ricardo, Pedrero, Denisa y Villegas. Espero no olvidarme de ninguno. También quiero agradecer su ayuda, soporte y compañía a los técnicos de laboratorio: Juanjo y Paco, grandes profesionales y buenas personas. Gracias a Carmen por su ayuda en todo lo referente a documentación necesaria durante el tiempo que he estado en el departamento de Arquitectura de Computadores. Vuestra compañía y ayuda han colaborado en la finalización de esta tesis.

Agradecer a Cristina, mi mujer y mi compañera de vida, su incondicional apoyo, ánimo y paciencia aportados para la consecución de mis objetivos y crecimiento personal. Igualmente, agradecer a mis padres su confianza y esfuerzo por darme la oportunidad de formarme en aquello que deseaba. A mis hermanos por enriquecer mi vida y saber que siempre estarán ahí. A mis amigos por ofrecerme distintos puntos de vista con los que afrontar las situaciones a las que nos enfrentamos.

Gracias a todos por haber estado ahí. Haber podido compartir todo este tiempo con vosotros me ha ayudado a aprender y adquirir herramientas útiles para todos los ámbitos de la vida.

Finalmente, mencionar las fuentes de financiación que han permitido que pue-
de llevar a cabo esta tesis: el proyecto TIN2016-80920-R del Gobierno de España
y al plan propio de la Universidad de Málaga.



UNIVERSIDAD
DE MÁLAGA

Resumen

En esta tesis se analiza el problema de planificar un conjunto de tareas sobre una GPU desde diferentes puntos de vista. Por una parte, se estudia el solapamiento de comandos de transferencia de datos con comandos de ejecución de *kernels* con el objetivo de minimizar el tiempo de ejecución (*makespan*). Por otra parte se comparan distintos métodos que permiten la ejecución solapada de varios *kernels* sobre la misma GPU buscando alcanzar diferentes objetivos como maximizar el rendimiento del sistema (*system throughput*), alcanzar la equidad (*fairness*) o garantizar una calidad de servicio (QoS).

En el estudio sobre el solapamiento de comandos se busca identificar el orden de ejecución que resulte en un tiempo de procesamiento mínimo. Se aplican los conceptos de la teoría de planificación a este problema y se modela la ejecución concurrente de tareas en una GPU como un problema de tipo *Flow Shop*. Además, se desarrolla una nueva estrategia llamada NEH-GPU que combina una heurística previamente existente con un modelo de ejecución de tareas en GPU y se efectúan experimentos para validar su eficacia y robustez.

En la tesis también se aborda el problema de la ejecución concurrente de *kernels* (CKE) analizándolo desde el punto de vista software y hardware. En este problema se busca planificar un conjunto de *kernels* para su coejecución y de esta forma mejorar el uso de los recursos hardware.

Nuestro modelo *software*, denominado *FlexSched*, implementa políticas de planificación destinadas a maximizar el rendimiento en la ejecución de los *kernels* o a satisfacer requisitos de calidad de servicio (QoS) de la misma, como por ejemplo el tiempo máximo de respuesta de un *kernel*. Una ventaja importante de *FlexSched* es que requiere solo modificaciones mínimas en el código del *kernel* y utiliza un *profiler on-line* productivo para lograr una distribución eficiente de los recursos de la GPU.

También se presenta un modelo *hardware*, HPSM (*Hybrid Piecewise Slowdown Model*), de planificación y ejecución concurrente de *kernels* en una GPU que permite mejorar el tiempo de ejecución de un conjunto de *kernels* y aplicar políticas orientadas al *fairness*. Este modelo puede predecir el progreso normalizado de los *kernels* y redistribuir la asignación de recursos para alcanzar los objetivos marcados.



UNIVERSIDAD
DE MÁLAGA

Índice general

Contenido	I
Lista de Figuras	V
Lista de Tablas	IX
1.- Introducción	1
1.1. Motivación	3
1.1.1. Concurrencia de comandos de transferencia con comandos de lanzamiento de <i>kernels</i>	3
1.1.2. Concurrencia de comandos de lanzamiento de <i>kernels</i>	5
1.2. Objetivos de la tesis	7
1.3. Estructura de la tesis	9
2.- Antecedentes	11
2.1. Buses de conexión	12
2.1.1. PCI Express (PCIe)	12
2.1.2. NVLink y NVSwitch	14
2.2. Arquitecturas NVIDIA	16
2.2.1. Kepler	17
2.2.2. Maxwell	21

2.2.3.	Pascal	22
2.2.4.	Volta	24
2.2.5.	Turing	26
2.2.6.	Ampere	27
2.3.	Interfaz de programación (API) CUDA	28
2.3.1.	CUDA	29
2.3.2.	Eventos y Profiling	36
2.4.	Planificador de Bloques y Warps	42
2.4.1.	Planificador de bloques	43
2.4.2.	Planificador de warps	44
2.5.	Simulador de GPUs	44
2.5.1.	Arquitectura global de GPGPU-Sim	45
2.5.2.	Componentes de GPGPU-Sim	46
SIMT Core Clusters	46	
Red de interconexión	46	
Partición de memoria	47	
2.6.	Métricas de rendimiento de un sistema	48
3.-	Ejecución solapada de transferencias con kernels	51
3.1.	Estado del Arte	52
3.1.1.	Modelado de Transferencias	52
3.1.2.	Modelado del tiempo de ejecución de Kernels	54
3.1.3.	Ejecución Concurrente de Comandos	55
3.2.	Lanzamiento Asíncrono de Comandos	55
3.3.	Estimación del Tiempo de Ejecución de Comandos	57
3.3.1.	Transferencias de Memoria	58
3.3.2.	Ejecución de Kernels	63
3.4.	Teoría de la Planificación	63

3.5. Heurísticas de Planificación	67
3.5.1. Slope index	67
3.5.2. NEH heuristic	68
3.5.3. Single queue	68
3.5.4. NEH heuristic para ejecución en GPU	69
3.6. Experimentos	71
3.6.1. Análisis estadístico	71
3.6.2. Aplicabilidad y Escalabilidad	73
3.6.3. Comparación con MPS	76
3.7. Resumen	78
4.- Ejecución concurrente de kernels mediante métodos software	81
4.1. Estado del Arte	82
4.2. Ejecución Concurrente de <i>Kernels</i> (CKE)	85
4.3. Motivación	88
4.4. FlexSched	90
4.5. Detalles de la implementación	92
4.5.1. Transformación del Kernel	92
4.5.2. Basic Scheduling Units (BSUs)	94
4.5.3. Configuración del espacio de coejecución	96
4.5.4. Profiling de la coejecución y planificación	97
4.6. Resultados experimentales	98
4.6.1. Coste la transformación del <i>kernel</i>	100
4.6.2. Retardo en el desalojo	101
4.6.3. Rendimiento de la planificación de CTAs	102
4.6.4. Rendimiento de la planificación <i>FlexSched</i>	105
4.6.5. Planificación orientada a la latencia con <i>FlexSched</i>	112
4.7. Resumen	113



5.- Planificación eficiente de kernels usando técnicas hardware	115
5.1. Estado del Arte	115
5.2. Motivación	118
5.3. Hybrid Slowdown Model (HSM)	122
5.4. Implementación <i>hardware</i>	127
5.5. Resultados experimentales	130
5.5.1. SMT vs SMK	131
5.5.2. Política <i>Fairness</i>	132
5.6. Resumen	135
6.- Conclusiones	137
6.1. Contribuciones y conclusiones	137
6.2. Aportaciones	140
6.3. Líneas futuras de investigación	143
6.3.1. Ejecución concurrente de <i>kernels</i> y transferencias de memoria	143
6.3.2. Aplicación de modelos predictivos	144
6.3.3. Procesamiento en memoria	144
Bibliografía	145

Índice de figuras

1.1. Sistema heterogéneo	2
1.2. Vista del <i>profiler</i> de la ejecución de 4 aplicaciones en una GPU . . .	4
1.3. IPC alcanzado por la ejecución individual de los <i>kernels</i> GCEDD y RED utilizando un número variable de CTAs mediante SMT . . .	6
1.4. STP alcanzado por la ejecución concurrente de los <i>kernels</i> GCEDD y RED utilizando un número variable de CTAs mediante SMT . . .	7
2.1. Conector NVLink entre dos GPUs	14
2.2. Conectores NVLink entre cuatro GPUs	15
2.3. Conector NVSwitch entre múltiples GPUs	15
2.4. SMX de la arquitectura Kepler de NVIDIA	20
2.5. Streams en arquitecturas Fermi (superior) y Kepler (inferior) . . .	22
2.6. SMM de la arquitectura Maxwell de NVIDIA	23
2.7. Arquitectura Pascal de NVIDIA	24
2.8. SM de la arquitectura Volta de NVIDIA	25
2.9. SM de la arquitectura Turing de NVIDIA	26
2.10. SM de la arquitectura Ampere de NVIDIA	28
2.11. Ejecución de 4 bloques de un programa CUDA en dos GPUs con 2 y 4 SMs, respectivamente	30
2.12. Jerarquía de acceso a memoria en CUDA	32
2.13. Arquitectura GPGPU-Sim	45
2.14. SIMT Core Cluster en GPGPU-Sim	46

2.15. Partición de memoria en GPGPU-Sim	48
3.1. Transferencia de datos	53
3.2. Esquema de lanzamiento en GPUs NVIDIA con dos motores DMA	57
3.3. Ancho de banda medio obtenido para las GPUs K20c, GTX 980 y Titan X usando transferencias de memoria <i>pinned</i> y paginable	60
3.4. Ancho de banda medio obtenido para las GPUs K20c, GTX 980 y Titan X con diferentes combinaciones de transferencias con memoria paginable y <i>pinned</i> en direcciones opuestas	61
3.5. Lanzamiento de tareas GPU como un problema <i>Flow Shop</i> de 3 máquinas	65
3.6. Diagrama de cajas de <i>speedup</i> sobre <i>makespan</i> medio de 10.626 combinaciones de 4 tareas en K20c, GTX 980 y Titan X	73
3.7. Proximidad al mejor resultado	75
3.8. Diagrama de cajas de <i>speedup</i> para las heurísticas NEH y NEH-GPU	77
3.9. Comparativa entre MPS y NEH GPU	78
4.1. Distribución simple de SMs	86
4.2. Distribución SMT de SMs	87
4.3. Distribución SMK de SM	87
4.4. Línea de temporal que muestra una visión general del enfoque <i>software</i> propuesto para la coejecución de <i>kernels</i>	89
4.5. Visión general de FlexSched	91
4.6. Planificación de tres BSUs en una GPU con cuatro SMs	95
4.7. FlexSched versus cCUDA	104
4.8. Inicio de <i>profiling</i> de <i>FlexSched</i>	106
4.9. Fin del <i>profiling</i> de <i>FlexSched</i>	108
4.10. <i>Profiling</i> de <i>FlexSched</i> para PF/SPMV	109
4.11. Coste de la fase de <i>profiling</i> y <i>speedup</i> frente a <i>HyperQ</i>	111
5.1. <i>IPC</i> y <i>STP</i> con RED y GCEDD	120
5.2. <i>Memoria DRAM</i>	122
5.3. Ancho de banda y <i>hit rate</i> en el <i>row buffer</i>	123



5.4. NP y Fairness para HSM-Fair aplicado a la coejecución SMK de RED y GCEDD 125

5.5. NP y Fairness para HPSM-Fair aplicado a la coejecución SMK de RED y GCEDD 126

5.6. Mejores resultados para ANTT y STP 132

5.7. Columna izquierda: *boxplots* del error en la predicción del *Fairness*, *ANTT* y *STP*. Columna derecha: resultados obtenidos con una política *fairness* de planificación de CTAs para las combinaciones ejecutadas 134



UNIVERSIDAD
DE MÁLAGA

Índice de tablas

2.1. Características de las versiones del bus PCIe	13
2.2. Características de las versiones del bus NVLink	16
2.3. Características de las versiones del bus NVSwitch	16
2.4. Características de las generaciones Kepler, Maxwell y Pascal de NVIDIA	18
2.5. Características de las generaciones Volta, Turing y Ampere de NVIDIA	19
2.6. Identificadores de funciones <i>kernel</i>	30
3.1. GPUs K20c, GTX980 y Titan X de NVIDIA	57
3.2. Parámetros del modelo de transferencia para las GPUs	62
3.3. Resumen de las notaciones para la teoría de la planificación	66
3.4. Tiempo de ejecución de los comandos de las tareas MM, BS, TM y VA	70
3.5. <i>Makespan</i> usando NEH y NEH-GPU	70
3.6. Tareas usadas en los experimentos CTE	71
4.1. Mejora de rendimiento SMT y SMK	88
4.2. Aplicaciones usadas en la validación del modelo <i>software</i>	99
4.3. Análisis de la transformación de los <i>kernels</i> para el modelo <i>software</i>	100
4.4. Porcentaje de CTAs por SM usados por los <i>kernels</i> no sensitivos a la latencia respecto al total de CTAs disponibles durante la eje- cución concurrente con el <i>kernel</i> CEDD	113

5.1. Valores HSM para las figuras 5.4a y 5.4b	125
5.2. Valores HPSM para las figuras 5.4a y 5.4b	127
5.3. <i>Kernels</i> usados en los experimentos para la validación del modelo <i>hardware</i>	131

1 Introducción

El auge de la inteligencia artificial (IA), el *big data* y el internet de las cosas (IoT) precisan de sistemas que permitan el procesamiento de grandes volúmenes de datos y realicen cálculos complejos a velocidades muy altas. Es por ello que las arquitecturas de computación de alto rendimiento (HPC) se han convertido en herramientas indispensables para el desarrollo de estas tecnologías y, por lo tanto, representan la base para los avances científicos, industriales y sociales. Los sistemas HPC más potentes, disponibles en la lista de los 500 mejores supercomputadores del mundo [90], están compuestos por CPUs de alto rendimiento, tarjetas de red, memorias para el almacenamiento de datos y aceleradores para cargas de trabajo especializadas. Los aceleradores más usados en estos sistemas son las GPUs [6, 58] (figura 1.1), mientras que otros sistemas incluyen FPGAs [102], aunque también existen circuitos integrados de aplicaciones específicas (ASIC) [9, 11], y arquitecturas de procesamiento en memoria (PIM) [2, 3, 21] que permiten optimizar ciertas aplicaciones de IA o trasladar el procesamiento a la memoria. Estos sistemas heterogéneos son usados concurrentemente por un gran número de usuarios que lanzan sus aplicaciones sobre estos sistemas para aprovechar sus capacidades de cómputo.

La heterogeneidad de elementos de procesamiento disponibles en los sistemas HPC permiten aprovechar la gran variedad de repertorios de instrucciones (ISA) que pueden ser utilizados para la optimización de algoritmos. La gestión de esta diversidad de procesadores requiere de un *software* que permita explotar al máximo el rendimiento de los *clústers* de HPC. Este *software* es el encargado de decidir las aplicaciones que serán ejecutadas en cada procesador, según criterios de productividad, consumo energético y/o reparto justo de los recursos. En la figura 1.1 se aprecia como las aplicaciones que llegan al servidor son lanzadas en



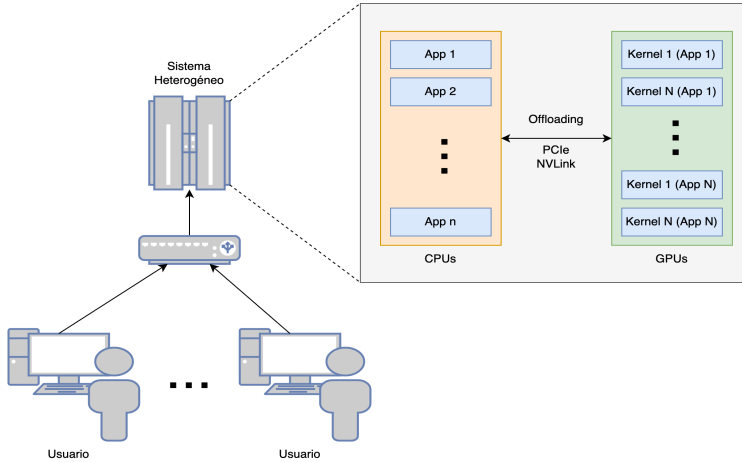


Figura 1.1: Sistema heterogéneo compuesto por CPUs y GPUs. Estos sistemas reciben peticiones de distintos usuarios para lanzar distintas aplicaciones con múltiples *kernels*.

las CPUs, mientras que partes del código de estas aplicaciones son lanzados en las GPUs. Las partes del código ejecutadas en las GPUs se denominan *kernels*.

La ejecución de aplicaciones secuenciales o moderadamente paralelas suelen ser apropiadas para CPU, mientras que las aplicaciones que realizan una computación masivamente paralela obtienen un mejor rendimiento en los aceleradores (GPUs, FPGAs o ASICs). El trabajo conjunto de distintas unidades de procesamiento requiere de una sincronización entre las mismas y una comunicación que permita compartir datos para la optimización de sus recursos. Estos dispositivos normalmente poseen espacios de memoria propios que deben comunicarse con la memoria de la CPU o *host* por medio de buses de interconexión (PCIe, NVLink o MXM entre otros) pudiendo provocar una alta penalización en el rendimiento. Es de suma importancia decidir el elemento de procesamiento adecuado para la ejecución de cada aplicación y, de este modo, mejorar el rendimiento general del sistema.

Las GPUs son uno de los aceleradores más utilizados en la actualidad para la ejecución de aplicaciones de alto rendimiento computacional. Los fabricantes de GPUs más relevantes son NVIDIA y AMD. Estos fabricantes han implementado entornos de desarrollo como CUDA [74], propio de NVIDIA, y OpenCL [42], una interfaz de programación de aplicaciones (API) que es la alternativa libre multiplataforma que han asumido distintos fabricantes. Estos entornos de desa-

rollo permiten aprovechar los recursos de cómputo de las GPUs. Además, son usados para planificar aplicaciones entre las distintas unidades de procesamiento que componen un sistema HPC, optimizando la ejecución de las aplicaciones y, por tanto, mejorando el rendimiento global del sistema.

1.1. Motivación

Las aplicaciones ejecutadas sobre sistemas heterogéneos están compuestas por distintas partes que son ejecutadas en el *host* (CPU) o en los dispositivos (GPU, FPGA, etc.). La ejecución de estas aplicaciones puede generar una infrautilización de los recursos disponibles dentro del sistema debida a la saturación de uno o varios de los recursos del sistema. Esto impide que el resto de recursos alcancen la ocupación máxima permitida. Con el fin de obtener un buen rendimiento del sistema y mejorar el uso de los recursos disponibles, se han propuesto una serie de técnicas de planificación de aplicaciones que permiten mejorar, entre otros, el *speedup* o la calidad de servicio (QoS). Estas técnicas se centran en la selección del orden de lanzamiento de las aplicaciones, y en la asignación de recursos a cada una.

1.1.1. Concurrencia de comandos de transferencia con comandos de lanzamiento de *kernels*

Una forma de conseguir la coejecución de aplicaciones en una misma GPU es mediante la planificación de comandos que puedan ser ejecutados concurrentemente. Estos comandos pueden ser de transferencia de datos entre el *host* y el *dispositivo*, en ambas direcciones, y de ejecución de *kernels*, los cuales son lanzados en unas colas denominadas *streams*. El orden en el que se lanza un conjunto de aplicaciones afecta al número de comandos que pueden ser ejecutados al mismo tiempo y, por tanto, a su tiempo de ejecución y rendimiento en la GPU. Por lo tanto, los *streams* pueden usarse para definir el orden de ejecución de los comandos que componen las distintas aplicaciones, ya que los comandos lanzados en un *stream* específico se ejecutan en orden de llegada. Es importante que, para cumplir la dependencia de comandos dentro de una misma aplicación, éstos se ejecuten por el mismo *stream*. Por otra parte, comandos lanzados por distintos *streams* pueden solapar su ejecución, dado que no existen dependencias entre comandos de distintos *streams* (salvo que se fuerce una dependencia de forma explícita). Finalmente, la ejecución concurrente de comandos pertenecientes a distintos *streams* depende de la disponibilidad de recursos *hardware* de la GPU.

La figura 1.2 muestra la salida gráfica de un *profiler* para visualizar el comportamiento de cuatro aplicaciones ejecutadas en una GPU. Por simplicidad, pero sin pérdida de generalidad, hemos supuesto que cada aplicación lanza sus comandos por un solo *stream*. La ejecución de varias aplicaciones suele seguir cierto orden: en primer lugar se lanzan una, varias o en algunos casos ninguna transferencia de memoria del *host* al *dispositivo* (*HtD*); en segundo lugar se ejecutan uno o más comandos de ejecución o *kernels*; finalmente una, algunas o ninguna transferencia de memoria del dispositivo al *host* (*DtH*). Las aplicaciones ejecutadas en la figura 1.2 se han seleccionado del conjunto de *benchmarks* del SDK de CUDA [66], del cual se han seleccionado *Matrix Multiplication* (MM), *Black Scholes* (BS), *Matrix Transposition* (TM) y *Vector Addition* (VA) lanzadas por los *streams* 13, 14, 15 y 16, respectivamente. Las cajas de colores cian (MM), azul marino (BS), magenta (TM) y púrpura (VA) representan la ejecución de cada *kernel*, mientras que las cajas marrones representan las transferencias entre el *host* y el *dispositivo* en ambas direcciones. Las transferencias antes de la ejecución del *kernel* son desde el *host* al *dispositivo* (*HtD*) y las transferencias tras el *kernel* son desde el *dispositivo* al *host* (*DtH*). El solapamiento de comandos de transferencia *HtD* y *DtH* se produce ya que la tarjeta discreta utilizada posee dos motores DMA (*Direct Memory Access*) que permiten transferencias de datos en sentidos opuestos de manera simultánea. Estos dos ejemplos se han ejecutado en una tarjeta K20c de NVIDIA. La imagen superior (*Permutation I*) se ha ejecutado con el orden MM-VA-TM-BS, mientras que en la imagen inferior (*Permutation II*) ha sido ejecutada con el orden BS-TM-MM-VA. Para la *Permutation II*, el cambio en el orden de ejecución de las aplicaciones permite la reducción del tiempo total de ejecución (*makespan*), influyendo positivamente en el uso del dispositivo.

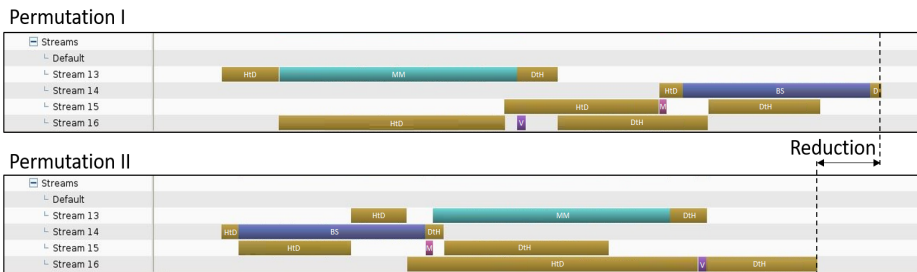


Figura 1.2: Vista del *profiler* de la ejecución de 4 aplicaciones en una GPU.

En el capítulo 3 de esta tesis se aborda el problema de la coejecución de aplicaciones en un entorno HPC con GPUs de NVIDIA conectadas por buses PCIe. Concretamente, se presentan ciertos mecanismos que permiten decidir, de

forma dinámica, el orden de ejecución de aplicaciones con el fin de solapar la ejecución de comandos de diferentes aplicaciones y maximizar el rendimiento en la GPU.

1.1.2. Concurrencia de comandos de lanzamiento de *kernels*

La ejecución de un *kernel* en una GPU puede no escalar adecuadamente debido a las restricciones de la arquitectura y a la saturación de recursos computacionales, por ejemplo por la saturación de los *pipelines* por dependencias RAW (en los *kernels* intensivos en cómputo) o el *trashing* de la caché L1 (en los *kernels* intensivos en L1) [103]. Los *kernels* intensivos en memoria pueden producir otro tipo de saturación, ya que pueden llegar a ocupar todo el ancho de banda de la memoria global antes de que se asignen todos los grupos de hilos (bloques de hilos o CTAs en CUDA) posibles a las unidades de cómputo del dispositivo (*Streaming Multiprocessors*, SMs, en terminología CUDA). En este caso, una forma de resolver el problema de saturación de recursos es reducir el número de CTAs asignados al *kernel* que produce la saturación en los SMs y, con el objetivo de mantener una alta utilización de la GPU, planificar simultáneamente los CTAs de otro *kernel* con necesidades de recursos diferentes. En la figura 1.3 se muestra la ejecución de un *kernel* intensivo en cómputo (GCEDD a la izquierda) y un *kernel* intensivo en memoria (RED a la derecha) siguiendo una distribución denominada SMT, donde en cada ejecución se incrementa el número de SMs asignados a cada aplicación hasta llegar al máximo de SMs existentes en la GPU. Con este experimento se pretende mostrar el efecto de la saturación en el rendimiento en términos de ejecución de instrucciones por ciclo (IPC) para cada *kernel*. El *kernel* RED satura la memoria antes de que se asignen todos los SMs, lo que limita su IPC y el rendimiento máximo que podría alcanzar si todos los SMs estuvieran asignados y no hubiera cuellos de botella en la memoria. Por el contrario, para el *kernel* GCEDD los valores de IPC son lineales conforme al número de SMs [32]. En este caso, el *kernel* lanza todos los CTAs en cada SM y el IPC obtenido en cada SM está limitado por el uso intensivo de los recursos compartidos dentro del SM.

La coejecución de estos dos *kernels* puede mejorar el rendimiento de la GPU. Al limitar el número de SMs asignados al *kernel* RED al número de SMs donde se produce la saturación de la memoria (y el IPC no aumenta), podrían asignarse los SMs libres al *kernel* GCEDD para comenzar su ejecución. La ejecución de dos *kernels* de forma concurrente puede producir interferencias entre ambos *kernels* al compartir los recursos de la GPU, lo que reduce el IPC alcanzado por cada uno, pero de forma global sí se puede conseguir una mejora. Por tanto, un objetivo

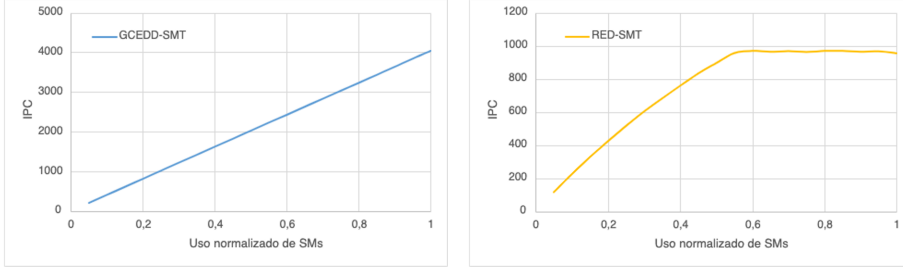


Figura 1.3: IPC alcanzado por la ejecución individual de los *kernels* GCEDD y RED utilizando un número variable de CTAs mediante SMT.

para este tipo de planificación podría ser maximizar el IPC alcanzado por la GPU.

El análisis de la coejecución de dos o más *kernels* requiere del uso de una métrica de rendimiento que mida el rendimiento global del sistema (*STP*). La expresión *STP* para un conjunto de *kernels* K coejecutados viene dada por la ecuación 1.1, donde IPC_k^{shared} e IPC_k^{alone} indican el *IPC* del *kernel* k cuando se coejecuta con otros *kernels* y el *IPC* del *kernel* k cuando se ejecuta en solitario, respectivamente. Cada término de la suma es conocido como progreso normalizado (NP), donde el *NP* de un *kernel* k viene dado por la ecuación 1.2.

$$STP = \sum_{k=1}^K \frac{IPC_k^{shared}}{IPC_k^{alone}} \quad (1.1)$$

$$NP_k = \frac{IPC_k^{shared}}{IPC_k^{alone}} \quad (1.2)$$

La figura 1.4 ilustra el *STP* alcanzado por cada *kernel* durante la coejecución de ambos (cuanto más alto, mejor). En el eje de abscisas se indica el valor normalizado de SMs lanzados por RED. GCEDD está utilizando los recursos restantes de la GPU para ocupar la mayor cantidad posible de SMs. El *STP* no sólo depende de los SMs asignados, sino también de las interferencias entre los *kernels* durante su coejecución causadas por el uso de otros recursos compartidos (unidades funcionales, colas de *load/store*, *row buffers* de memoria global, etc.). Así, siguiendo una distribución SMT, en el que todos los recursos de un SM se asignan a un mismo *kernel*, la interferencia aparece a nivel de la memoria global, ya que los *kernels* coejecutados van a competir por los recursos de memoria global.

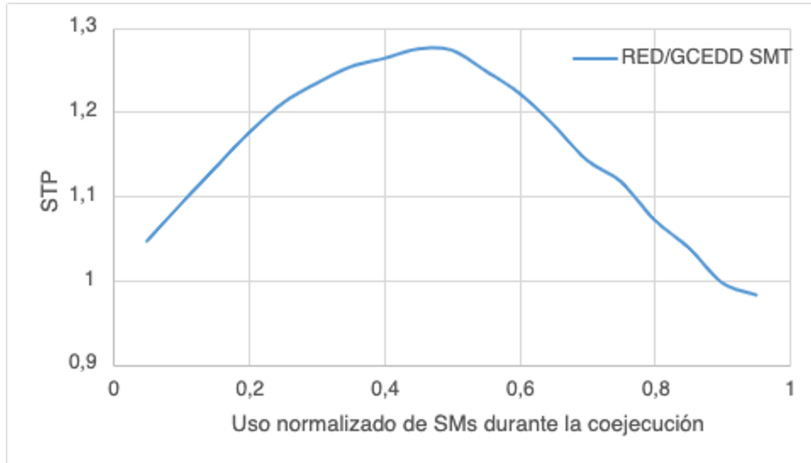


Figura 1.4: STP alcanzado por la ejecución concurrente de los *kernels* GCEDD y RED utilizando un número variable de CTAs mediante SMT.

También se puede observar que el *STP* toma valores muy diferentes en función del número de SMs que han sido ejecutados por cada *kernel*. Así, los valores de *STP* oscilan entre 0,98 a 1,27 para una distribución SMT. La asignación de SMs que obtiene un mayor IPC no se conoce de antemano, ya que depende de la interferencia entre los *kernels* al utilizar recursos compartidos como la memoria global, las cachés L1 y L2, las colas de *load/store*, las unidades funcionales, etc.

Por tanto, puede resultar ventajoso co-ejecutar varios *kernels* para compartir los recursos de una GPU, y es importante determinar que asignación de recursos permite obtener el rendimiento máximo.

1.2. Objetivos de la tesis

Teniendo en cuenta los dos ejemplos motivadores mostrados en la sección anterior, vamos a definir los objetivos de este trabajo de investigación. Así, nos hemos planteado estudiar la ejecución concurrente de comandos en una GPU, tanto la coejecución de comandos de transferencia con comandos de lanzamiento de *kernels*, como la coejecución simultánea de varios *kernels*, con el propósito de mejorar distintos aspectos de la planificación, como son el rendimiento (*system throughput*, STP), la equidad en el reparto de recursos (*fairness*) o para garantizar

una calidad de servicio (*quality of service, QoS*). El desarrollo de estos objetivos permitirá realizar aportaciones en tres líneas diferentes:

- Ejecución solapada de comandos de transferencia con *kernels*
 - Obtención de un modelo de ejecución de aplicaciones que permita simular con exactitud los procesos de transferencia de datos, simultáneas o solas, y la ejecución de *kernels* en tarjetas discretas que utilizan buses de interconexión con las CPUs.
 - Implementación de un sistema de ejecución de aplicaciones que permita explotar las posibilidades de concurrencia entre comandos pertenecientes a aplicaciones distintas.
 - Propuesta de una heurística que permita encontrar el mejor orden de ejecución dentro de un conjunto de aplicaciones aplicando políticas para reducir el tiempo de ejecución global.
 - Validación de la heurística propuesta con cargas de trabajo real, que permitan representar cualquier combinación de grupos de aplicaciones, en las cuales se hagan un uso diverso de los recursos con el fin de ilustrar una amplia variedad de comportamientos.
- Ejecución concurrente de *kernels* con métodos *software*
 - Propuesta de un mecanismo *software* que permita gestionar, en tiempo de ejecución, la asignación de recursos de forma adecuada entre dos *kernels* a coejecutar, dentro de un conjunto de *kernels*.
 - Implementación del mecanismo propuesto para que sea posible redistribuir de forma eficiente los recursos entre los *kernels*.
 - Validación del mecanismo *software* propuesto con aplicaciones reales, con una amplia variedad de ellas para representar comportamientos diversos, y cumpliendo con los objetivos de planificación.
- Planificación eficiente de *kernels* usando técnicas *hardware*
 - Propuesta de un mecanismo *hardware* capaz de encontrar, en tiempo real, la distribución adecuada de los recursos *hardware* de la GPU.
 - Implementación de dicho mecanismo en un simulador avanzado, GPGPU-Sim [12], usando políticas de distribución SMT y SMK.
 - Validación del mecanismo *hardware* con *kernels* reales, aplicando políticas de planificación tanto de minimización del tiempo de ejecución como orientadas a un reparto justo (*fairness*) de recursos.

1.3. Estructura de la tesis

Las arquitectura más recientes de las GPUs de NVIDIA, junto a los buses de conexión existentes, se describen en el capítulo 2, analizando sus características de rendimiento. También se describe la API de CUDA para la implementación de aplicaciones que permitan ejecutar parte de sus instrucciones en la GPU, y el trabajo que realiza el planificador de bloques e hilos de NVIDIA. Además, en este capítulo se describen los eventos de CUDA y el uso de ellos para el *profiling* de aplicaciones. También se detalla en este capítulo uno de los simuladores de GPUs de NVIDIA más avanzados, GPGPU-Sim [12], que será usado en la propuesta de mecanismos de planificación *hardware* para la GPU. Finalmente, se introducen las métricas de rendimiento usadas en esta tesis.

En el capítulo 3 se desarrollan los métodos aplicados a la ejecución solapada de transferencias con *kernels* en una GPU y se estudia la planificación dinámica de las tareas mediante una heurística para encontrar el mejor orden de ejecución. En primer lugar, se describen los comandos que componen una aplicación ejecutada en una GPU y se realiza un estudio de las aplicaciones más usadas en el estado del arte para proponer un modelo que permita predecir el tiempo de ejecución de las transferencia de datos, tanto en solitario como solapadas con otra transferencia en sentido contrario, y el tiempo de ejecución de los *kernels*. Además, se detalla el lanzamiento de comandos por *streams* para permitir la coejecución de comandos. La selección de un orden de ejecución adecuada requiere de una heurística de planificación apropiado como la propuesta en esta tesis para la ejecución solapada de comandos. Todo ello es validado mediante experimentos realizados usando nuestro modelo de planificación.

El problema de la ejecución concurrente de *kernels* (CKE) aplicando métodos *software* se aborda en el capítulo 4. En él se estudia la distribución de los recursos *hardware* de cómputo de la GPU mediante técnicas *software* con el fin de reducir el tiempo de ejecución de un conjunto de *kernels*. Los componentes de la arquitectura de nuestro mecanismo, *FlexSched*, se detallan para ilustrar la función que ejerce cada componente dentro del sistema. Finalmente, se valida nuestro mecanismo *software* mediante una serie de experimentos.

La ejecución concurrente de *kernels* mediante la aplicación de técnicas *hardware* es estudiada en el capítulo 5. En este caso se implementa un sistema de lanzamiento de *kernels* siguiendo las estrategias SMT y SMK, al que se le añade un mecanismo *hardware* para determinar el progreso normalizado de cada *kernel*. Nuestro modelo, *HPSM-Fair*, encuentra la mejor planificación, desde el punto de vista de la equidad, de los recursos *hardware* entre dos *kernels*. También se

presenta un estudio de los resultados obtenidos que permiten validar la utilidad del modelo propuesto.

Para terminar, las conclusiones obtenidas en esta tesis se puntualizan en el capítulo 6, con un resumen de las aportaciones más relevantes de esta tesis junto con las publicaciones que se han llevado a cabo. Finalmente, se proponen una serie de líneas de investigación futuras basadas en el trabajo presentado en esta tesis.

2 Antecedentes

El uso de aceleradores como las GPUs y las arquitecturas MIC (*Many Integrated Core*, p.ej., Intel Xeon Phi) ha dado lugar al desarrollo de entornos de programación que permitan aprovechar las características de estos aceleradores. Los entornos de programación más relevantes para este tipo de procesadores son NVIDIA-CUDA [74] y OpenCL [42].

La capacidad para reducir el tamaño de los transistores ha facilitado el incremento del número de transistores que se pueden integrar en un mismo chip. Esto ha permitido desarrollar CPUs que integran procesadores gráficos (GPUs) en un único procesador y aumentar las capacidades de cómputo de las GPUs discretas. Los procesadores heterogéneos resuelven las limitaciones al compartir datos, ya que la memoria de ambos procesadores es la misma. Existen diferentes fabricantes que han lanzado al mercado este tipo de arquitecturas integradas, algunos de ellos son AMD Kaveri [7], NVIDIA Tegra [65] e Intel con sus procesadores de séptima generación Kaby Lake [34]. En estas arquitecturas se pueden producir algunos inconvenientes como el aumento de los retardos al acceder a memoria o la disminución de las capacidades de cómputo en sus procesadores gráficos. Es por ello que en esta tesis nos centramos en el uso de las GPUs discretas de NVIDIA, las cuales también se están beneficiando de los avances tecnológicos, aumentando sus capacidades de cómputo y disminuyendo su consumo.

En este capítulo se dará una visión general de las GPUs discretas. Los buses de conexión con sus características y rendimiento se describen en la sección 2.1. La sección 2.2 muestra la evolución de las arquitecturas de NVIDIA y las mejoras que han ido introduciendo cada una de ellas hasta la actualidad. La API de CUDA es presentada en la sección 2.3, incluyendo algunas aplicaciones de *profiling* y de uso de eventos de CUDA. El planificador de bloques e hilos de NVIDIA se

describe en la sección 2.4, mientras que en la sección 2.5 se describe GPGPU-Sim, el simulador de GPU más utilizado en trabajos de investigación. Finalmente, se incluye una introducción a las métricas usadas en esta tesis en la sección 2.6.

2.1. Buses de conexión

Las tarjetas gráficas discretas de NVIDIA soportan distintos buses de conexión, según el formato de la tarjeta y la comunicación que se quiere realizar entre los dispositivos que conforman el sistema.

El bus de conexión más común usado para comunicarse con el *host* del sistema o CPU es el puerto *PCI Express* (PCIe). Los sistemas móviles usan tarjetas gráficas con un factor de forma especial, por lo que poseen un bus de interconexión denominado *Mobile PCI Express Module* (MXM).

Las tecnologías NVLink y NVSwitch permiten mejorar la velocidad de comunicación entre tarjetas gráficas, optimizando la escalabilidad de cargas de trabajo HPC e IA.

2.1.1. PCI Express (PCIe)

El puerto *PCI Express* (PCIe) es un bus serie de datos de alta velocidad. La funcionalidad del puerto PCIe es comunicar entre sí los distintos componentes *hardware* de cualquier equipo o sistema HPC. Surgió con el fin de sustituir a los buses PCI y AGP. Las mejoras proporcionadas por este bus son respecto al rendimiento, consumo de energía y los mecanismos para la detección e informe de errores en la transmisión de información.

La principal mejora respecto a los buses anteriores es que el ancho de banda del bus no se divide entre los distintos componentes *hardware*, sino que se comparte. En el bus PCI se utiliza una arquitectura de bus paralelo, donde el *host* y el resto de dispositivos comparten un conjunto de direcciones, datos y líneas de control. Esto limita la comunicación entre el *host* y un dispositivo en un momento determinado y en una sola dirección. Además, la frecuencia de reloj del bus PCI viene limitada por la frecuencia de reloj del dispositivo más lento. Por otra parte, en PCIe la topología es punto a punto y los enlaces serie conectan cada dispositivo al *host* de forma individual, permitiendo una comunicación *full-duplex* entre los componentes. De esta forma todos los dispositivos son capaces de recibir el ancho de banda disponible sin restricciones dadas por otros dispositivos.

Desde su lanzamiento en 2003, se han desarrollado cinco versiones distintas del bus PCIe. Estas versiones están resumidas en la tabla 2.1, sus características son las siguientes:

- **PCIe 1.0:** se presentó en 2003, con una capacidad de transferencia de datos de 2,5 GT/s (*GigaTransfers* por segundo) y un ancho de banda de 250 MB/s por bus de datos.
- **PCIe 2.0:** lanzada en 2007, la transferencia de datos pasa a 5 GT/s y el ancho de banda por bus de datos es de 500 MB/s
- **PCIe 3.0:** se comercializa en 2010, con una tasa de transferencia de 8 GT/s y 984,6 MB/s de ancho de banda por bus de datos.
- **PCIe 4.0:** se lanza en 2017, doblando la transferencia a 16 GT/s y 1.969 MB/s de ancho de banda por bus de datos.
- **PCIe 5.0:** presentada en 2021 con una transferencia de 32 GT/s y 3.938 MB/s de ancho de banda por bus de datos.

Versión	Capacidad de transferencia	Ancho de banda
1.0	2,5 (GT/s)	250 (MB/s)
2.0	5 (GT/s)	500 (MB/s)
3.0	8 (GT/s)	984,6 (MB/s)
4.0	16 (GT/s)	1.969 (MB/s)
5.0	32 (GT/s)	3.938 (MB/s)

Tabla 2.1: Características de las versiones del bus PCIe. Se muestran sus capacidades de transferencia de datos en GT/s y su ancho de banda por vía de datos en MB/s.

Una de las particularidades del bus PCIe es que es un bus modular. Esto se debe a que los conectores a la placa no tienen todos las mismas características y depende del número de vías de datos conectadas al puerto, lo que permite aumentar su ancho de banda. Físicamente, cada conector posee un número de pines de contacto determinado, según el número de pines habilitados pueden ser x1, x4, x8 o x16.

Existen otros tipos de PCIe. Uno de ellos está basado en fibra óptica que se usa para comunicar distintos dispositivos *hardware*, comúnmente GPUs, dentro de un servidor, ya que permite un gran ancho de banda. También existe el bus *Mobile PCI Express Module* (MXM). Este bus de conexión ha sido diseñado para usar

GPUs en portátiles usando el estándar PCIe. El objetivo es crear un estándar no propietario. Esto permite integrar tarjetas gráficas en equipos de tamaño reducido y posibilita el intercambio por otro modelo, sin la necesidad de cambiar todo el sistema. Existen dos generaciones distintas de los módulos MXM, y estas dos generaciones no son compatibles entre ellas. Ambas permiten integrar cualquier modelo de GPU, aunque el tamaño máximo del módulo está limitado. En 2012 el bus de conexión MXM 3.1 agregó soporte a PCIe 3.0.

2.1.2. NVLink y NVSwitch

El uso de la IA y HPC ha incrementado la necesidad de más *hardware* que permita mejorar el rendimiento y las características de los sistemas que los soportan. Esto ha generado una gran demanda de sistemas con múltiples GPUs que actúan conjuntamente como un gran acelerador. El ancho de banda de los buses PCIe es reducido para este tipo de tareas, por lo que se crean cuellos de botella en la comunicación entre tarjetas gráficas. El nacimiento de los buses NVLink y NVSwitch surge para comunicar múltiples GPUs de forma más eficiente y rápida, sin generar cuellos de botella en la comunicación con el host.

NVIDIA NVLink es un bus de conexión directa entre dos GPUs a velocidades elevadas (varios ejemplos de esquemas de conexión con este bus se muestran en las figuras 2.1 y 2.2). Por otro lado, NVIDIA NVSwitch incorpora varios NVLinks que comunican distintas GPUs a la misma velocidad que un bus NVLink; este esquema de conexión se puede apreciar en la figura 2.3.

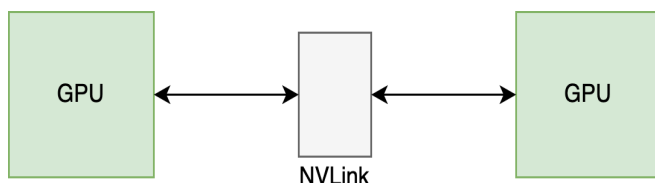


Figura 2.1: Conector NVLink entre dos GPUs.

Las GPUs de NVIDIA A100 *Tensor Core* admiten hasta 12 conexiones NVLink de tercera generación pudiendo alcanzar un ancho de banda de 600 GB/s, lo que implica hasta 6 veces más del ancho de banda del PCIe en su quinta generación. Las características de las generaciones del bus NVLink se muestran en la tabla 2.2.

El incremento en la información a manejar por los sistemas de IA y HPC ha

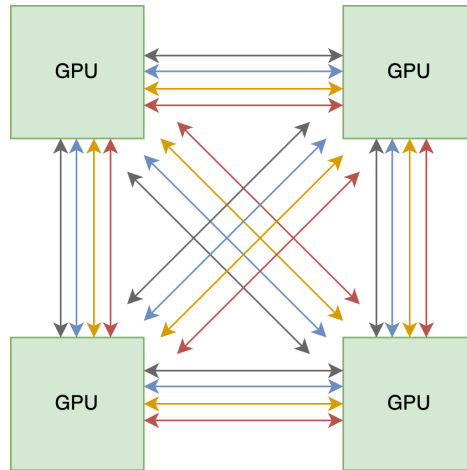


Figura 2.2: Conectores NVLink entre cuatro GPUs.

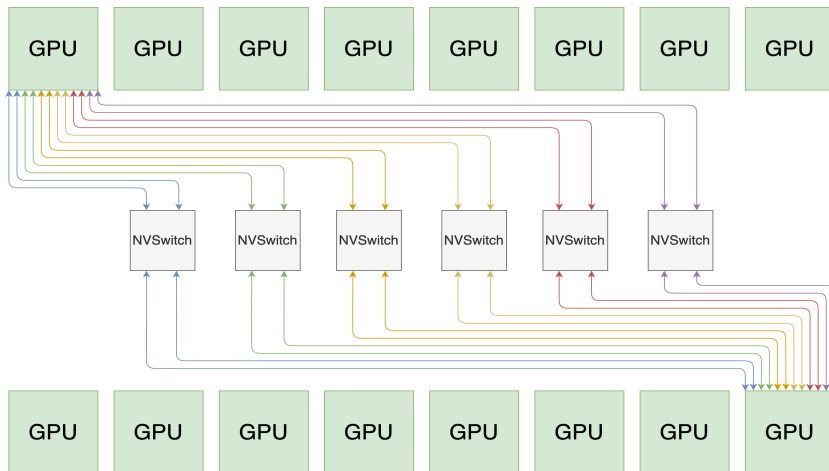


Figura 2.3: Conector NVSwitch entre múltiples GPUs.

producido que la comunicación entre tarjetas GPUs sea llevada a cabo de forma más rápida y escalable. Para hacer estas cargas de trabajo más escalables, el ancho de banda debe ser considerablemente mayor y tener una latencia reducida.

NVSwitch está basada en la potencia de conexión de los buses NVLink. Esto crea un tejido de GPUs concentradas en un único servidor. Este bus NVSwitch permite conectar de 8 a 16 GPUs en un único servidor y habilita la comunicación

	1 ^a Generación	2 ^a generación	3 ^a generación
Ancho de banda	160 (GB/s)	300 (GB/s)	600 (GB/s)
Enlaces máximos	4	6	12
Arquitectura NVIDIA	Pascal	Volta	Ampere

Tabla 2.2: Características de las versiones del bus NVLink [73]. Se muestra el ancho de banda alcanzado en (GB/s), el número máximo de enlaces máximos que permite cada GPU y la arquitectura que lo implementa.

directa punto a punto entre todas las GPUs, permitiendo usar hasta 16 GPUs como un acelerador único con espacio de memoria unificada y hasta 5 petaFLOPs de potencia de computación aplicado a IA. Las características de las distintas generaciones en las que se pueden integrar los buses de interconexión NVSwitch se muestran en la tabla 2.3.

	1 ^a Generación	2 ^a generación	3 ^a generación
Número de GPUs	Hasta 8	Hasta 16	Hasta 16
Ancho de banda GPU a GPU	160 (GB/s)	300 (GB/s)	600 (GB/s)
Ancho de banda	1,28 (TB/s)	4,8 (TB/s)	9,6 (TB/s)
Arquitectura NVIDIA	Pascal	Volta	Ampere

Tabla 2.3: Características de las versiones del bus NVSwitch [73]. Se muestra el número máximo de GPUs que pueden ser conectadas de forma directa, el ancho de banda alcanzado en (GB/s) de GPU a GPU en el NVSwitch, el ancho de banda máximo alcanzado por la comunicación entre todas las GPUs y la arquitectura que los soporta.

La incorporación de NVLinks en el bus de conexión NVSwitch hace que sean componentes esenciales dentro la gran variedad de soluciones disponibles en los centros de procesamiento de datos donde existen GPUs de NVIDIA. Esto habilita a los investigadores para ofrecer soluciones a problemas reales de forma escalada e incrementando la aceleración de estos sistemas a gran escala.

2.2. Arquitecturas NVIDIA

Los fabricantes de GPUs más utilizados, en sistemas de IA y HPC, son los fabricantes NVIDIA, AMD e Intel. En esta sección se describen las arquitecturas que ha presentado NVIDIA, desde la arquitectura Kepler hasta la arquitectura Ampere, incidiendo en las plataformas con la que se ha estado trabajando, defi-

niendo sus características específicas y la innovación que se ha ido introduciendo en cada generación.

NVIDIA es uno de los principales referentes en el diseño de GPUs. En el año 2007 NVIDIA lanzó *Compute Unified Device Architecture* (CUDA), una plataforma de computación paralela que permite a los programadores acelerar sus algoritmos codificando porciones de su código para ser ejecutadas en una GPU. CUDA explota las ventajas de la GPU frente a la CPU, que permite el paralelismo ofrecido por sus múltiples núcleos, lanzando un gran número de hilos de forma simultánea. Esto convierte a las GPUs de NVIDIA en una de las alternativas más importantes en la computación de altas prestaciones.

En noviembre de 2006 NVIDIA lanzó su primera generación de GPUs orientadas a la ejecución de aplicaciones de propósito general (GPGPU). Ocho han sido las arquitecturas que han ido introduciendo importantes avances de una generación a otra, gracias en parte a la integración de cada vez más transistores. Las arquitecturas que ha lanzado NVIDIA hasta inicios de 2022 son: Tesla, Fermi, Kepler, Maxwell, Pascal, Volta, Turing y Ampere. En esta tesis nos hemos centrado en las arquitecturas Kepler, Maxwell, Pascal y Volta, aquellas con las que se han realizado nuestros experimentos. Las características generales de las arquitecturas propuestas por NVIDIA desde la generación Kepler hasta la actualidad se resumen en las tablas 2.4 y 2.5. Las tablas muestran también un número de versión asignado por NVIDIA, denominado capacidad de computación (Compute Capability), formado por dos términos X.Y. El valor de X se corresponde con la arquitectura del núcleo de computación, y el valor de Y con una revisión menor de la arquitectura.

2.2.1. Kepler

En 2012, NVIDIA lanza la arquitectura Kepler [67]. Ésta mejoró la eficiencia energética ya que bajó la frecuencia de reloj. Esto no redujo su rendimiento gracias a que el proceso de fabricación pasó a ser de 28 nm. Con esta reducción NVIDIA pudo mejorar el diseño de los SMs Kepler, mostrados en la figura 2.4. Los SMs pasan a ser conocidos como SMX, y contienen 196 núcleos de procesamiento. Cada SM posee 4 planificadores de *warps* soportando la ejecución de hasta 4 *warps* distintos al mismo tiempo. De esta forma los SMXs de Kepler consiguieron doblar el rendimiento de los SMs de la arquitectura Fermi.

La arquitectura Kepler introduce el paralelismo dinámico. Este permite a la GPU generar más trabajo sobre sí misma, lo que incrementa la potencia de cómputo de la arquitectura. Por otra parte, el flujo de trabajo en la arquitectura

	Kepler (GK110B)	Maxwell (GM107)	Pascal (GP102)
Compute Capability	3.5	5.0	6.1
SMs	15	16	20
Núcleos/SM	192	128	128
Planif. de Warps/SM	4	4	4
Mem. Compartida/SM (KB)	64	96	96
Registros/SM	65536	65536	65536
Hilos/Warp	32	32	32
Hilos/Bloque	1024	1024	1024
Bloques/SM	16	16	32
Reloj Núcleos (MHz)	705	1033	1303
Computación FP32 (TFLOPs)	5,046	6,688	11,76
Computación FP64 (TFLOPs)	1,682	2,229	0,367
Reloj VRAM (GHz)	1,75	1,253	1,808
Ancho de banda (GB/s)	336 GDDR5	320 GDDR5	694,3 GDDR5X
Kernel Concurrentes	Si	Si	Si
Solapamiento de Comandos	Si	Si	Si
Proceso litográfico (nm)	28	28	16
Año de Lanzamiento	2012	2014	2016

Tabla 2.4: Características de las generaciones Kepler, Maxwell y Pascal de NVIDIA.

	Volta (GV100)	Turing (TU102)	Ampere (GA102)
Compute Capability	7.0	7.5	8.6
SMs	84	72	84
Núcleos/SM	128	64	128
Planif. de Warps/SM	4	4	4
Mem. Compartida/SM (KB)	128	96	128
Registros/SM	65536	65536	65536
Núcleos Tensores/SM	2	8	4
Hilos/Warp	32	32	32
Hilos/Bloque	1024	1024	1024
Bloques/SM	32	32	32
Reloj Núcleos (MHz)	1245	1395	1395
Computación FP32 (TFLOPs)	16,35	16,32	35,58
Computación FP64 (TFLOPs)	8,177	0,509	0,556
Computación Tensor (TFLOPs)	118,5	113,8	119
Reloj VRAM (GHz)	1,106	1,75	1,219
Ancho de banda (GB/s)	1133	672	936,2
	HBM2	GDDR6	GDDR6X
Kernel Concurrentes	Si	Si	Si
Solapamiento de Comandos	Si	Si	Si
Proceso litográfico (nm)	12	12	8
Año de Lanzamiento	2018	2018	2020

Tabla 2.5: Características de las generaciones Volta, Turing y Ampere de NVIDIA.

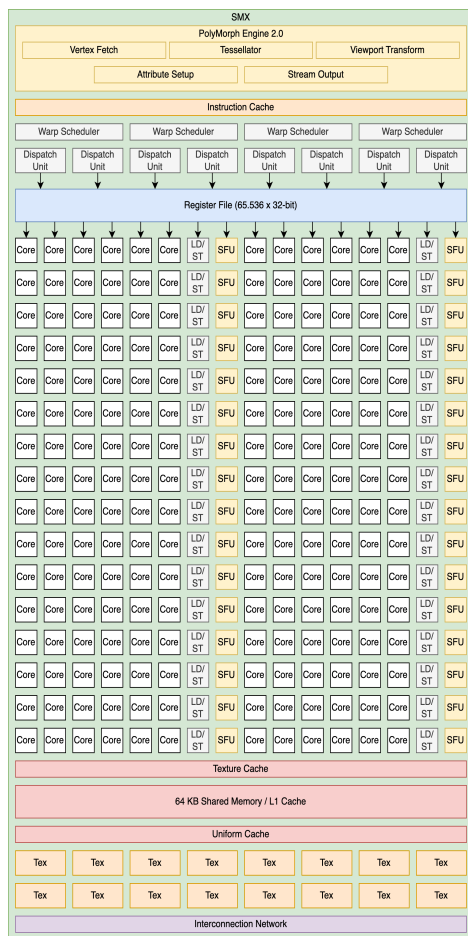


Figura 2.4: SMX de la arquitectura Kepler de NVIDIA.

Fermi era unidireccional desde el *host* a la GPU por medio de la unidad CWD (*CUDA Work Distributor*). Ésto impide la ejecución de bloques de distintos *grids* al mismo tiempo. Kepler mejora su comportamiento por medio de GMU (*Grid Management Units*) que al comunicarse con CWD permite mantener *grids* listos para ser ejecutados, siendo capaz de lanzar hasta 32 *grids* distintos. Los *grids* son un conjunto de bloques de hilos. La comunicación entre CWD y GMU es bidireccional, permitiendo a la GMU pausar el lanzamiento de nuevos *grids*, mantenerlos pendientes o suspenderlos.

El rendimiento de una GPU puede venir limitado por cuellos de botella en memoria o por divergencias dentro de los *warps*. Esto causa que los núcleos CUDA estén infrautilizados. Esta infrautilización puede resolverse por medio de la coejecución de distintas aplicaciones o *kernels* dentro de una GPU con el uso de los *streams*. Un *stream* es una cola que almacena, en orden, los comandos que van a ser ejecutados. El lanzamiento de comandos por distintos *streams* puede permitir la coejecución de distintas aplicaciones si todas las aplicaciones en coejecución tienen recursos libres donde ejecutarse. Esta coejecución podría incrementar el rendimiento general de la GPU. En la arquitectura Fermi se pueden lanzar hasta 16 comandos de forma concurrente desde diferentes *streams*, aunque todos esos comandos finalmente son multiplexados en una única cola de trabajo. Esto genera una falsa dependencia en este *stream* disminuyendo la capacidad de ejecución concurrente en la GPU. Sin embargo, la arquitectura Kepler introduce *Hyper-Q*, el cual incrementa el número de colas de ejecución entre el *host* y la GPU, permitiendo hasta 32 *streams*. Las dependencias entre las colas de trabajo son optimizadas, por lo que las operaciones dentro de cada *stream* no bloquean la ejecución del resto de *streams*. La figura 2.5 muestra un ejemplo en la que se puede observar el comportamiento de los *streams* en ambas arquitecturas. En ambos casos, existen dos *streams* donde se lanzan siete tareas distintas de la siguiente manera: A, B, C y D en el *stream* 1, y E, F y G en el *stream* 2. En la arquitectura Fermi, al insertar cada *stream* sus tareas en un único *stream*, las dos únicas aplicaciones que podrían ser ejecutadas de forma concurrente serían D y E. Por otro lado, en la arquitectura Kepler, al no existir dependencias entre *streams* gracias a *Hyper-Q*, se asocia una cola *hardware* diferente a cada *stream* y, por lo tanto, las tareas de los dos *streams* podrían ejecutarse concurrentemente.

Por último, se introduce también *Multi-Process Service* (MPS), un servicio que usa las características de *Hyper-Q* para ejecutar *kernels* de diferentes procesos de forma simultánea en una misma GPU, lo que puede mejorar el rendimiento cuando las capacidades de cómputo de la GPU están infrautilizadas por un solo *kernel* en ejecución.

2.2.2. Maxwell

Las novedades introducidas en la arquitectura Maxwell [69] en el año 2014 se centran en la eficiencia energética, ya que es una generación orientada a los PCs reducidos y portátiles.

La idea principal fue abandonar los SMX de Kepler, proponiendo los *Stream Maxwell Multiprocessor* (SMM), mostrados en la figura 2.6, que reducen el número

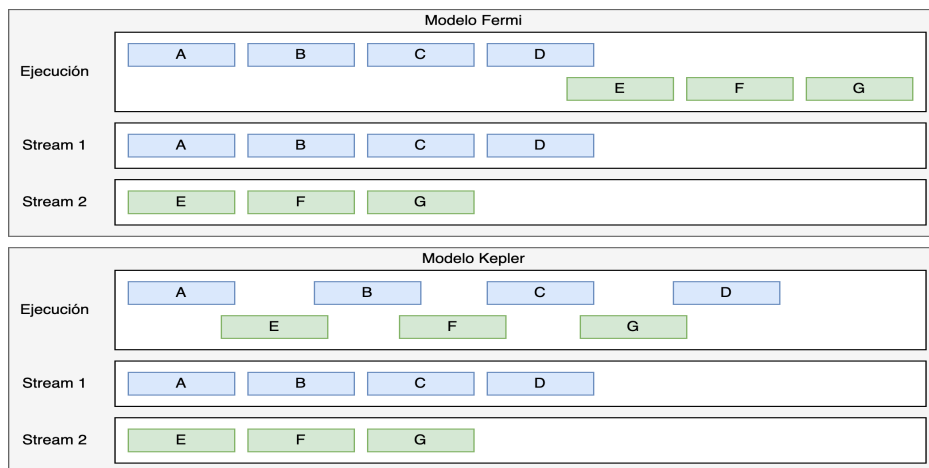


Figura 2.5: Streams en arquitecturas Fermi (superior) y Kepler (inferior). En la arquitectura Fermi solo D y E pueden ejecutarse concurrentemente dada a la dependencia causada por el uso de una sola cola *hardware*. En la arquitectura Kepler, utilizando Hyper-Q, las parejas de *kernels* que pueden coejecutarse aumentan gracias a que se usan múltiples colas *hardware*.

ro de núcleos dentro del SM a 128. Ahora, cada uno de los cuatro planificadores de *warps* trabajan únicamente para 32 núcleos. Ajustar el número de núcleos al tamaño de *warps* mejora la estructura del procesador, reduciendo su espacio y mejorando el consumo energético. Esta arquitectura mantiene las *Hyper-Q* que permiten generar trabajo desde la GPU. La caché L2 se incrementa a 2 MB, lo que reduce la comunicación con la memoria y, por tanto, se puede mantener un rendimiento similar con un ancho de banda menor. De esta forma, el bus de memoria pasa a ser de 128 bits, con el fin de reducir la energía consumida.

2.2.3. Pascal

La arquitectura que llegó a continuación fue la arquitectura Pascal [70] en el año 2016. El SM de la arquitectura Pascal es el mismo que el de la arquitectura Maxwell (figura 2.6). El proceso de fabricación de 16 nm mejoró sustancialmente su rendimiento, incrementando el número de SMs en un mismo chip. La arquitectura Pascal se muestra en la figura 2.7. Esta arquitectura cuenta con un total de 4 GPCs (clusters de procesamiento gráfico), donde cada GPC contiene a su vez 5 SMs, lo que suman un total de 20 SMs.

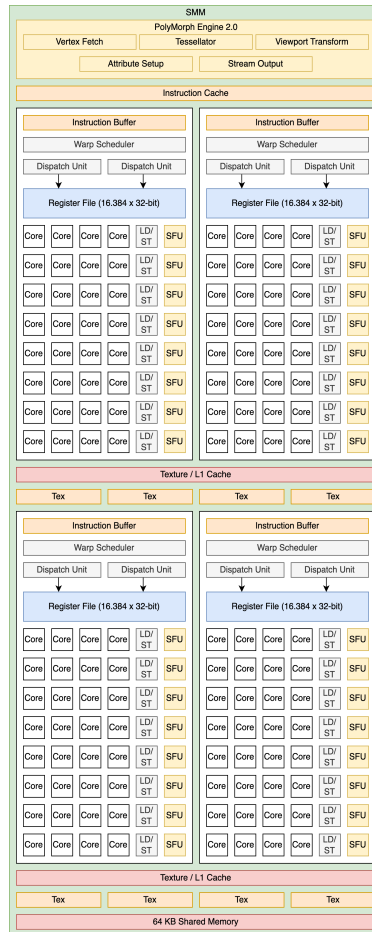


Figura 2.6: SMM de la arquitectura Maxwell de NVIDIA.

Además, se introdujo la memoria GDDR5X que proporciona velocidades de transferencia de hasta 10 Gbps gracias a sus ocho controladoras de memoria y su interfaz de 256 bits proporciona hasta un 43 % más en el ancho de banda respecto a la arquitectura Maxwell. La arquitectura Pascal también incorpora Hyper-Q y la unidad GMU.

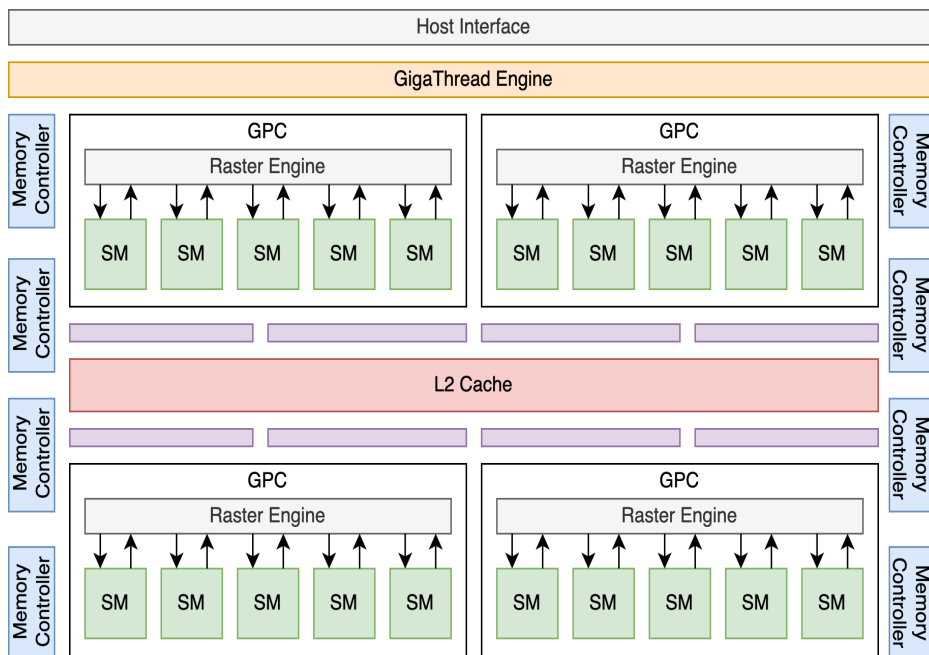


Figura 2.7: Arquitectura Pascal de NVIDIA.

2.2.4. Volta

La arquitectura Volta [71], introducida en 2017, incrementa el número de GPCs a 6, donde cada uno de ellos posee 14 SMs. Esta arquitectura cambia la composición de los SMs permitiendo la ejecución simultánea de instrucciones de enteros y flotantes, gracias a sus *cores* dedicados para ambas operaciones. Además, con Volta aparecen por primera vez los *Tensor Cores*. Los *Tensor Cores* mejoran las operaciones con matrices, optimizando el entrenamiento de redes neuronales, y permiten alcanzar hasta 125 TFLOPs para el entrenamiento y la inferencia de aplicaciones de inteligencia artificial. La arquitectura del SM Volta se ilustra en la figura 2.8.

La tecnología empleada en la memoria es HBM2. La tecnología HBM2 usa cuatro capas por *stack* de memoria, lo que permite un tamaño máximo de 16 GB de memoria interna de la GPU. HBM2 alcanza un ancho de banda máximo de 900 GB/s.

Por último, la arquitectura Volta introduce nuevas funciones MPS respecto a

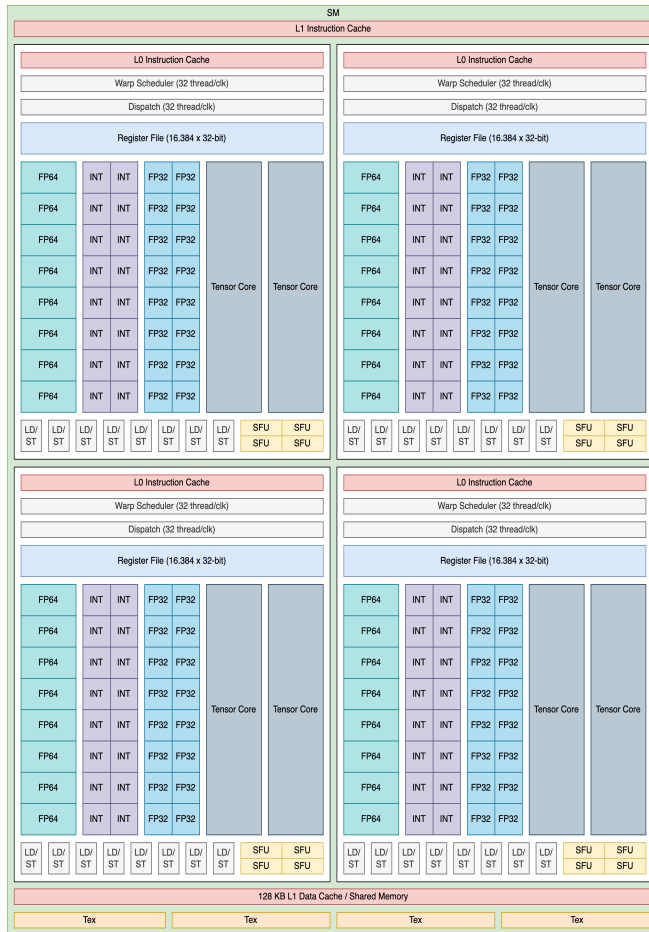


Figura 2.8: SM de la arquitectura Volta de NVIDIA.

las generaciones previas:

- Los clientes MPS pueden enviar sus trabajos directamente a la GPU sin pasar por el servidor MPS.
- Cada cliente MPS posee su propio espacio de direcciones en la GPU en lugar de compartirlo con el resto de clientes.
- Volta permite el aprovisionamiento de recursos para garantizar una calidad de servicio (QoS).

2.2.5. Turing

La arquitectura Turing [72] fue presentada en 2018 y supuso el mayor salto arquitectónico en más de una década. La arquitectura utiliza unos nuevos SMs (figura 2.9) que incluyen *hardware* dedicado para *Ray Tracing*, y además se mantienen los *Tensor Cores*.

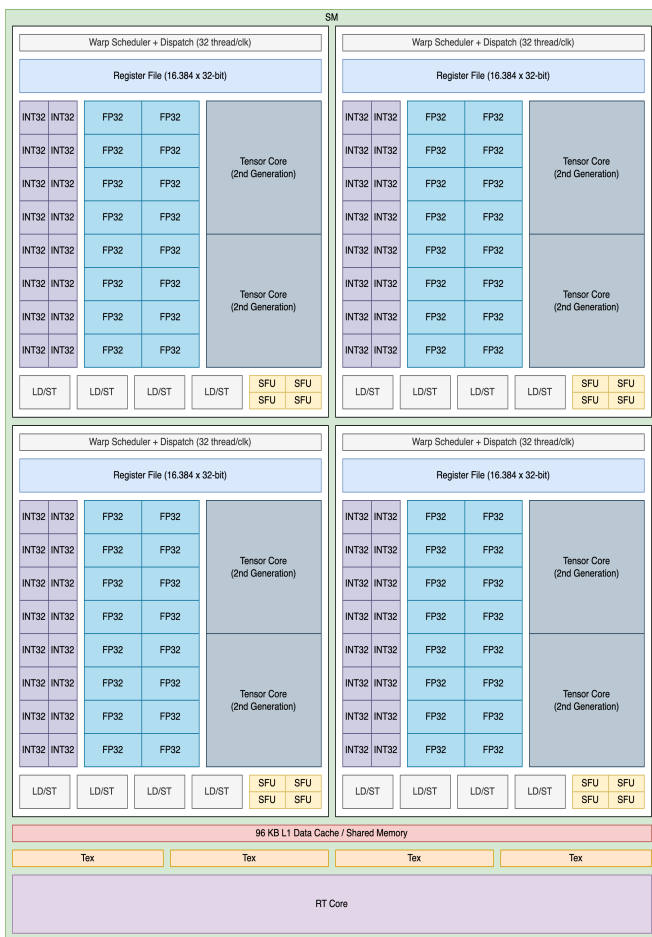


Figura 2.9: SM de la arquitectura Turing de NVIDIA.

Las tres nuevas características que introduce la arquitectura Turing son:

- El núcleo CUDA es escalable siendo capaz de ejecutar instrucciones de

enteros y flotantes en paralelo, al igual que en la arquitectura Volta.

- Uso del sistema de memoria GDDR6 con 16 controladoras, alcanzando hasta 14 Gbps.
- Los hilos no comparten punteros de instrucciones en los *warps*.

La introducción de la programación de hilos independientes en la arquitectura Volta hace que cada hilo tenga su propio contador de programa y los SMs puedan ejecutar hilos en un *warp* sin la necesidad de esperar a que converjan el resto de hilos dentro del mismo *warp*.

2.2.6. Ampere

La arquitectura Ampere [68] se lanza en 2020, introduciendo ciertos cambios en los SMs de la arquitectura Turing. Los SMs, mostrados en la figura 2.10, utilizan nuevos *Tensor Cores* que permiten operar con datos en punto flotante de hasta 64 bits, incluyendo números en formato BFLOAT (Brain Float Point), además de enteros y datos con precisión mixta.

En esta generación se introduce un nuevo avance que permite instanciar múltiples GPUs dentro de una misma GPU. A este avance se le conoce como *Multi-Instance GPU* (MIG) [59]. Esta tecnología permite crear hasta siete instancias independientes dentro de una misma GPU, cada una con su ancho de banda a memoria, caché y *cores* de cómputo. Esto permite al administrador de los recursos la posibilidad de gestionar cualquier carga de trabajo, desde cargas de trabajo pequeñas a cargas de trabajo más elevadas, garantizando la calidad de servicio (QoS) y permitiendo el acceso a los recursos de cómputo de la GPU a todos los usuarios del sistema. Al depender de la asignación de recursos que realice el administrador del sistema para instanciar las distintas GPUs dentro de una sola GPU, el reparto a realizar de los recursos de cómputo deben seguir una distribución *spatial multitasking* (SMT) limitando las ventajas que se pueden obtener con un reparto *simultaneous multikernel* (SMK). Estas distribuciones se describen con más detalle en la sección 4.2. Además, son posibles situaciones en la que ciertos recursos de la GPU queden ociosos y, por tanto, se pierda capacidad de cómputo para aplicaciones que pudieran ejecutarse más rápidamente si se les asignase los recursos ociosos en dicho momento, o incluso que durante la ejecución de distintas aplicaciones, en instancias diferentes, interfieran en el rendimiento de forma desfavorable al competir por los recursos de la GPU.



Figura 2.10: SM de la arquitectura Ampere de NVIDIA.

2.3. Interfaz de programación (API) CUDA

Existe gran variedad de interfaces y lenguajes de programación para arquitecturas GPU y MIC, siendo los más utilizados CUDA [74] y OpenCL [42]. En esta sección se describe la interfaz de programación CUDA empleada para gestionar los recursos *hardware* de las GPUs de NVIDIA (subsección 2.3.1) y, además, algunas aplicaciones de *profiling* y de uso de eventos de CUDA (subsección 2.3.2).

2.3.1. CUDA

El lenguaje de programación que permite codificar algoritmos en la GPUs de NVIDIA se llama CUDA. Sus siglas provienen de *Compute Unified Device Architecture* (Arquitectura Unificada de Dispositivos de Computo). Esto hace referencia tanto a su compilador, como a la API desarrollada por NVIDIA para poder ejecutar programas de propósito general en sus GPUs. Esta API permite desarrollar código mediante una extensión del lenguaje de programación C [15].

En CUDA existen tres abstracciones relevantes: 1) una jerarquía de grupos de hilos, 2) un espacio de memoria compartida entre los hilos y 3) barreras de sincronización. Estas abstracciones proporcionan paralelismo a bajo nivel, tanto de datos como de hilos, todo ello dentro de un paralelismo a alto nivel con el paralelismo de tareas. Para ello el programador debe dividir el problema en tareas más pequeñas que puedan ser resueltas de forma concurrente por medio de bloques de hilos. A su vez, estas tareas más pequeñas pueden dividirse para ser resueltas usando subconjuntos de hilos dentro del mismo bloque.

Los bloques de hilos pueden ser planificados en cualquier *Streaming Multi-processor* (SM) de la GPU, sin ningún orden, de forma concurrente o secuencial. Esto permite escalar la ejecución de un programa CUDA, ya que puede ser ejecutado en cualquier número de SMs independientemente de la GPU donde va a ser ejecutado. En la figura 2.11 se muestra un ejemplo de ejecución de un *kernel* que ejecuta 4 bloques de hilos en dos GPUs distintas, donde la GPU 0 tiene 2 SMs, mientras que la GPU 1 tiene 4 SMs. Suponiendo que los SMs de ambas GPUs tienen la misma capacidad de computo, el tiempo de ejecución de cada bloque de hilos debe ser el mismo en las dos GPUs, por lo tanto, la GPU 1 al tener el doble de SMs puede asignar más recursos para la ejecución concurrente de todos los bloques que ejecuta el *kernel* tardando la mitad de tiempo en ejecutar todos los bloques de hilos que la GPU 0 con 2 SMs. Esta característica de CUDA permite escalar la aplicación en GPUs distintas, sin la necesidad de realizar modificaciones en el código.

Los *kernels* son las funciones declaradas en C que pueden ser ejecutadas en una GPU. Estas funciones tienen la característica de ser ejecutadas N veces por M hilos CUDA de forma paralela. Las funciones *kernels* son definidas con los identificadores reservados `__global__` (líneas 2 y 3 del código 2.1) o `__device__`. Ambas funciones *kernel* son ejecutadas en el dispositivo, pero mientras que las funciones con identificador `__global__` se lanzan desde el *host*, las funciones con el identificador `__device__` se lanzan desde el propio dispositivo tal y como se ilustra en la tabla 2.6. El número de hilos CUDA usados para su ejecución se declaran en su llamada de la siguiente forma: `<<<N, M>>>` (línea 15 del código

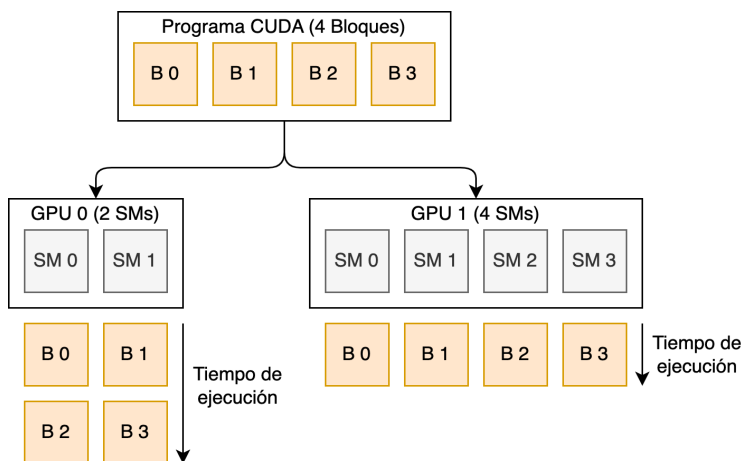


Figura 2.11: Ejecución de 4 bloques de un programa CUDA en dos GPUs con 2 y 4 SMs, respectivamente.

2.1), donde N se corresponde con el número de bloques necesarios para ejecutar un *kernel* y M es el número de hilos que hay dentro de cada bloque. Ambos parámetros pueden ser representados por un entero (*int*) o el tipo tridimensional de CUDA (*dim3*). El número de bloques N que ejecuta dicho *kernel* se denomina *grid*. Los bloques pueden tener 1, 2 o 3 dimensiones. Un hilo dentro de un bloque de hilos es identificado por el valor de *threadIdx*, el bloque al que pertenece el hilo viene dado por *blockIdx*, y la dimensión de cada bloque se accede por medio de la variable *blockDim*. De esta forma, el índice de cada hilo dentro del *grid* se puede calcular dentro del propio *kernel* por medio del identificador de bloque (*blockIdx.x*), la dimensión del mismo (*blockDim.x*) y el identificador del thread (*threadIdx.x*) (línea 4 del código 2.1). En el caso del código 2.2, en el que se multiplican dos matrices, los hilos se agrupan en dos dimensiones por lo que el identificador i del hilo se calcula por medio de la dimensión x (línea 4 del código 2.2), mientras que el identificador j se calcula con la dimensión y (línea 5 del código 2.2).

Identificador	Llamada desde	Ejecutada en
<code>__global__</code>	host	dispositivo
<code>__device__</code>	dispositivo	dispositivo

Tabla 2.6: Identificadores de funciones *kernel*. Origen de la llamada y lugar de ejecución.

```

1  /* Definición del kernel */
2  __global__ void vectorAddition(float *A, float *B, float *C,
3                               int nElem) {
4      int i = blockIdx.x * blockDim.x + threadIdx.x;
5
6      if (i < nElem)
7          C[i] = A[i] + B[i];
8  }
9
10 int main() {
11     /* Reserva de memoria e iniciación de vectores A, B y C */
12     ...
13
14     /* Llamada al kernel */
15     vectorAddition<<<N, M>>>(A, B, C, nElem);
16
17     /* Liberación de memoria y finalización */
18     ...
19 }

```

Código 2.1: Función CUDA para sumar dos vectores con N bloques de M hilos cada uno de ellos.

```

1  /* Definición del kernel */
2  __global__ void matrixMultiplication(float **A, float **B,
3                                     float **C, int nFil, int nCol)
4  {
5      int i = blockIdx.x * blockDim.x + threadIdx.x;
6      int j = blockIdx.y * blockDim.y + threadIdx.y;
7
8      if (i < nFil && j < nCol)
9          C[i][j] = A[i][j] + B[i][j];
10 }
11
12 int main() {
13     /* Reserva de memoria e iniciación de las matrices A, B y C */
14     ...
15
16     /* Definición de dimensiones de número de hilos por bloque */
17     dim3 nHilos(16, 16);
18     /* Definición de dimensiones de bloques en el grid */
19     dim3 nBloques(nFil/nHilos.x, nCol/nHilos.y);
20
21     /* Llamada al kernel */
22     matrixMultiplication<<<nBloques, nHilos>>>(A, B, C, nFil, nCol);
23
24     /* Liberación de memoria y finalización */
25     ...
26 }

```

Código 2.2: Función CUDA para multiplicar dos matrices

Los hilos de un bloque pueden cooperar entre sí compartiendo información por medio de la memoria compartida. Esta cooperación se produce cuando los accesos a memoria se coordinan, y para ello los hilos deben sincronizarse. La sincronización de los hilos de un bloque se lleva a cabo por medio de la función de CUDA `__syncthreads()`, en la que todos los hilos de un bloque esperan hasta que el resto de hilos hayan llegado a esta función para continuar con la ejecución del *kernel*.

Existen distintos espacios de memoria a los que acceden los hilos durante la ejecución del *kernel*. Estos espacios de memoria se ilustran en la figura 2.12. Los hilos poseen un espacio de memoria local accesible de forma privada por cada hilo. Los bloques pueden acceder a un espacio de memoria compartida accesible únicamente por los hilos del mismo bloque. Finalmente, todos los hilos pueden acceder a un espacio de memoria global de la GPU que se encuentra compartida por todos los hilos.

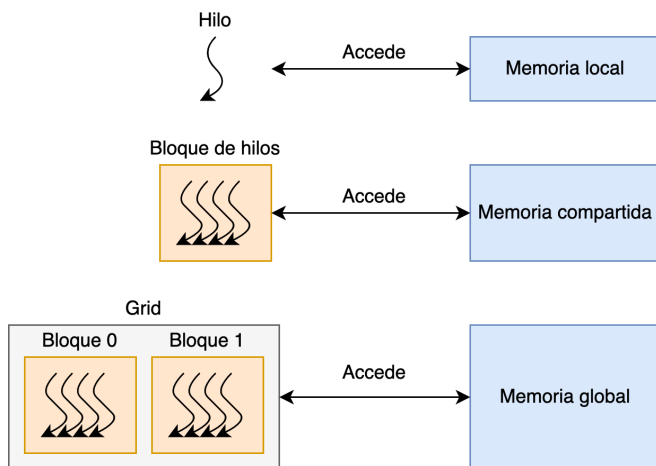


Figura 2.12: Jerarquía de acceso a memoria en CUDA.

La ejecución de un *kernel* en una GPU de NVIDIA requiere de una preparación previa a su ejecución y una posterior liberación de recursos. Los pasos a seguir para este proceso se ilustran en el código 2.3. Este ejemplo muestra los pasos necesarios para ejecutar el *kernel* del código 2.1 (*vectorAddition*):

1. Selección de la GPU (línea 9 del código 2.3).
2. Reserva de memoria (líneas 12-14 del código 2.3) e inicialización de vectores (líneas 21-22 del código 2.3) en el *host*.

3. Reserva de memoria en el dispositivo o GPU (líneas 16-18 del código 2.3).
4. Transferencia de información desde el *host* al dispositivo (líneas 24-25 del código 2.3).
5. Lanzamiento del *kernel* (líneas 28-30 del código 2.3).
6. Transferencia de resultados obtenidos desde el dispositivo al *host* (línea 33 del código 2.3).
7. Liberación de memoria en el dispositivo y en el *host* (líneas 36-37 del código 2.3, respectivamente).

Los comandos ejecutados en una GPU (lanzamiento de *kernels* y transferencias con la memoria) son lanzados en *streams*. Los *streams* son secuencias de comandos lanzados por el *host* de forma ordenada en la GPU. Todas las operaciones lanzadas por el mismo *stream* mantienen su orden de lanzamiento, mientras que las lanzadas por distintos *streams* pueden intercalarse e incluso ejecutarse de forma concurrente si existen recursos *hardware* suficientes en la GPU para su ejecución simultánea. El programador es el encargado de la selección de los *streams* donde se van a ejecutar los distintos comandos. Si el programador no selecciona ningún *stream*, el *stream* seleccionado es el *stream* por defecto (*default stream*). El *default stream* es el único que se sincroniza con el resto de operaciones lanzadas en la GPU. Por ello, toda operación lanzada en el *default stream* debe esperar a la finalización del resto de operaciones lanzadas en la GPU por distintos *streams*, y además ninguna operación lanzada por cualquier *stream* comenzará su ejecución hasta que las operaciones previas lanzadas en el *default stream* hayan finalizado.

Con la versión 7 de CUDA se introduce el *flag* `cudaStreamNonBlocking` con el cual se puede evitar la sincronización producida por el *default stream*. En el ejemplo del código 2.3 se lanzan todos los comandos por el *default stream*, de esta forma todos los comandos lanzados en la GPU son ejecutados en orden secuencial. Los comandos de transferencia (`cudaMemcpy`) son síncronos, mientras que los lanzamientos de *kernels* son asíncronos. Al tener una función de transferencia síncrona (líneas 24-25 del código 2.3), el *host* no lanza el *kernel* hasta la finalización de la transferencia de memoria desde el *host* al dispositivo. La transferencia de memoria del dispositivo al *host* (línea 33 del código 2.3) no se produce hasta la finalización del *kernel*, ya que se encuentran dentro del mismo *stream* (*default stream*).

A partir de la versión 6 de CUDA se introduce la técnica de memoria unificada. Esta técnica permite realizar las transferencias de los datos desde el *host* al dis-

positivo conforme el propio *kernel* vaya solicitándolos. Para ello, la reserva de memoria se realiza por medio de la función `cudaMallocManaged(&A, vectorSize)`, donde *A* es el puntero al vector y *vectorSize* el tamaño de dicho vector.

```

1 int main()
2 {
3     int nElem = 2048;
4     float *h_A, *d_B, *d_C;
5     float *d_A, *d_B, *d_C;
6     int vectorSize = nElem * sizeof(float);
7
8     /* Seleccion de GPU */
9     cudaSetDevice(0);
10
11    /* Reserva de memoria en el host */
12    h_A = (float *) malloc(vectorSize);
13    h_B = (float *) malloc(vectorSize);
14    h_C = (float *) malloc(vectorSize);
15    /* Reserva de memoria en el dispositivo */
16    cudaMalloc((void **)&d_A, vectorSize);
17    cudaMalloc((void **)&d_B, vectorSize);
18    cudaMalloc((void **)&d_C, vectorSize);
19
20    /* Inicializacion de vectores en el host */
21    inicializarVector(h_A, nElem);
22    inicializarVector(h_B, nElem);
23    /* Transferencia de informacion desde host al dispositivo */
24    cudaMemcpy(d_A, h_A, vectorSize, cudaMemcpyHostToDevice);
25    cudaMemcpy(d_B, h_B, vectorSize, cudaMemcpyHostToDevice);
26
27    /* Lanzamiento del kernel */
28    dim3 nHilos(1024, 1, 1);
29    dim3 nBloques(nElem/nHilos.x, 1, 1);
30    vectorAddition<<<nBloques, nHilos>>>(d_A, d_B, d_C, nElem);
31
32    /* Transferencia de resultado del dispositivo al host */
33    cudaMemcpy(h_C, d_C, vectorSize, cudaMemcpyDeviceToHost);
34
35    /* Liberacion de memoria */
36    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
37    free(h_A); free(h_B); free(h_C);
38 }

```

Código 2.3: Pasos para la ejecución de un kernel.

El programador puede definir *streams* (*non-default streams*), que son creados y destruidos desde el *host*. En este caso, para realizar transferencias de memoria se debe usar la función no bloqueante `cudaMemcpyAsync` donde se añade un último parámetro a la función con el identificador del *stream* (líneas 28-31 y 40-

41 del código 2.4). En el lanzamiento de un *kernel* por un *stream* se le deben pasar dos nuevos argumentos: el tamaño de la memoria compartida reservada dinámicamente por el *kernel* y el identificador del *stream* (línea 36 del código 2.4). El uso de *streams* añade dos nuevos pasos al proceso de ejecución de *kernels*: un paso previo para crear los *streams* y uno final donde se liberan los *streams* reservados. En el código 2.4 se muestran todos los pasos necesarios para ejecutar el *kernel* *vectorAddition* del código 2.1 usando un *stream*. Las llamadas a estas funciones desde el punto de vista del *host* son asíncronas, mientras que para la GPU, al lanzarse por un mismo *stream*, son síncronas.

En algunas situaciones es necesario sincronizar el *host* con los *streams* en el dispositivo. Esta sincronización se hace por medio de la función *cudaDeviceSynchronize* (línea 44 del código 2.4). En el caso en el que solo se quiera saber desde el *host* si las operaciones lanzadas por un *non-default streams* han terminado, se hace con la función *cudaStreamQuery(stream)*.

```

1  int main()
2  {
3      int nElem = 2048;
4      float *h_A, *d_B, *d_C;
5      float *d_A, *d_B, *d_C;
6      int vectorSize = nElem * sizeof(float);
7      cudaStream_t stream;
8
9      /* Creacion del stream */
10     cudaStreamCreate(&stream);
11
12     /* Seleccion de GPU */
13     cudaSetDevice(0);
14
15     /* Reserva de memoria en el host */
16     h_A = (float *)malloc(vectorSize);
17     h_B = (float *)malloc(vectorSize);
18     h_C = (float *)malloc(vectorSize);
19     /* Reserva de memoria en el dispositivo */
20     cudaMalloc((void **)&d_A, vectorSize);
21     cudaMalloc((void **)&d_B, vectorSize);
22     cudaMalloc((void **)&d_C, vectorSize);
23
24     /* Inicializacion de vectores en el host */
25     inicializarVector(h_A, nElem);
26     inicializarVector(h_B, nElem);
27     /* Transferencia de informacion desde host al dispositivo */
28     cudaMemcpyAsync(d_A, h_A, vectorSize, cudaMemcpyHostToDevice,
29                    stream);
30     cudaMemcpyAsync(d_B, h_B, vectorSize, cudaMemcpyHostToDevice,
31                    stream);
32

```

```

33  /* Lanzamiento del kernel */
34  dim3 nHilos(1024, 1, 1);
35  dim3 nBloques(nElem/nHilos.x, 1, 1);
36  vectorAddition<<<<nBloques, nHilos, 0, stream>>>(d_A, d_B, d_C,
37                                                  nElem);
38
39  /* Transferencia de resultado del dispositivo al host */
40  cudaMemcpyAsync(h_C, d_C, vectorSize, cudaMemcpyDeviceToHost,
41                stream);
42
43  /* Sincronizacion del host con la GPU */
44  cudaDeviceSynchronize();
45
46  /* Liberacion de memoria */
47  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
48  free(h_A); free(h_B); free(h_C);
49
50  /* Destruccion del stream */
51  cudaStreamDestroy(stream);
52 }

```

Código 2.4: Pasos para la ejecución de un kernel en un stream.

2.3.2. Eventos y Profiling

Las GPUs de NVIDIA pueden ser monitorizadas por medio de una serie de herramientas. Estas herramientas proporcionan información de la ejecución final de las aplicaciones, como pueden ser *NVIDIA Visual Profiler* [62], *nvprof* [63], *nsight* [61] y *CUPTI* [57].

Estas herramientas son capaces de monitorizar el comportamiento de la GPU para cada comando lanzado en la misma, proporcionando información relativa a las transferencias de memoria, ejecución de los *kernel*, colas de ejecución (*streams*) por las que se lanzan los comandos, información del tiempo de ejecución de los comandos, el número de bloques lanzados por un *kernel*, el número de hilos usados en cada bloque y un amplio catálogo de información [64]. Además, *CUPTI* (*CUDA Profiling Tools Interface*) permite la creación de herramientas de *profiling* mediante distintas APIs:

- API de actividad: registra de forma asíncrona las actividades que realiza CUDA, por ejemplo, la ejecución de los *kernels* o las copias de memoria.
- API de *callbacks*: es un mecanismo de callback de eventos CUDA que notifica al suscriptor que un evento de CUDA específico ha sido ejecutado, por ejemplo, el inicio de la copia a memoria, en tiempo de ejecución.

- API de eventos: recoge los contadores de rendimiento de los *kernels* durante la ejecución de los mismos.
- API de métricas: proporciona métricas de rendimiento de los *kernels* durante la ejecución de los mismos.
- API de profiling: recolecta las métricas de rendimiento durante un tiempo determinado.
- API de muestreo: reúne datos de muestreo de forma continua sin serializar la ejecución del *kernel*.
- API de puntos de control: proporciona soporte para guardar y restaurar automáticamente el estado funcional del dispositivo CUDA.

La información que recolectan las distintas APIs de CUPTI conllevan un alto coste en tiempo, por lo que el uso de estas APIs durante la ejecución del *kernel* es adecuado para estudiar el comportamiento y uso de recursos de una aplicación, pero no para aplicar políticas de planificación en tiempo real.

Además de monitorizar en tiempo de ejecución los comandos lanzados, estas herramientas permiten observar el comportamiento de la GPU. La gestión de las herramientas es llevada a cabo desde el *host* por medio del uso de eventos asíncronos de CUDA. Estos eventos son insertados por parte del *host* en cualquier punto de ejecución del programa con el fin de consultar el estado de ejecución del comando o comandos requeridos. La finalización de un evento ocurre cuando todas las tareas o comandos, anteriores a la ocurrencia del evento en el *stream* seleccionado, han terminado. Al igual que la gestión del evento se realiza desde el *host*, la creación y destrucción del mismo también se hace desde el *host*. En el código 2.5 se muestra un ejemplo en el que se usan los eventos CUDA para medir el tiempo de ejecución de un *kernel*. En primer lugar, es necesario definir las variables de eventos CUDA (línea 3 del código 2.5). La creación de los eventos se refleja en las líneas 9 y 10 del código 2.5. La inserción del evento *inicio* toma el tiempo en el que se empieza a medir la duración del comando en la línea 16 del código 2.5. Todos los comandos introducidos tras esta línea y antes de la inserción del evento *fin* en la línea 23 del código 2.5 entrarán dentro del tiempo de ejecución a medir. En este caso se ha medido el tiempo de ejecución de un *kernel* (líneas 19-20 del código 2.5), pero podría medirse el tiempo de ejecución de cualquier comando introducido en el *stream* 0. Al introducirse el comando de finalización, se espera a su finalización con un evento de sincronización en la línea 26 del código 2.5. En el caso de querer consultar el estado del evento sin bloquear la ejecución del *host*, se puede usar la función `cudaEventQuery(fin)` que

permitiría pasar al siguiente comando. En este caso, como se quiere hacer una medida de tiempo, es necesario bloquear su ejecución hasta su finalización. Tras ello, se calcula el tiempo transcurrido entre la inserción de los eventos *inicio* y *fin* con el comando de la línea 29 del código 2.5. Finalmente, se destruyen los eventos creados en las líneas 36 y 37 del código 2.5.

```

1  int main() {
2      cudaEvent_t inicio , fin;
3      float tiempo_ejecucion;
4
5      /* Inicializacion de vectores y creacion del stream */
6      ...
7
8      /* Creacion de los eventos */
9      cudaEventCreate(&inicio);
10     cudaEventCreate(&fin);
11
12     /* Transferencia de informacion desde host al dispositivo */
13     ...
14
15     /* Insercion del evento de inicio en el stream 0 */
16     cudaEventRecord(inicio , 0);
17
18     /* Lazamiento del kernel en el stream 0 */
19     vectorAddition<<<<nBloques , nHilos , 0 , stream>>>(d_A, d_B, d_C,
20                                                         nElem);
21
22     /* Insercion del evento de fin en el stream 0 */
23     cudaEventRecord(fin , 0);
24
25     /* Esperar a la finalizacion del evento fin */
26     cudaEventSynchronize(fin);
27
28     /* Asignacion de tiempo de ejecucion a variable */
29     cudaEventElapsedTime(&tiempo_ejecucion , inicio , fin);
30
31     /* Transferencia de informacion desde dispositivo al host ,
32        liberacion de memoria y destruccion del stream */
33     ...
34
35     /* Destruccion de los eventos */
36     cudaEventDestroy(inicio);
37     cudaEventDestroy(fin);
38 }

```

Código 2.5: Uso de eventos para la medición de tiempos de ejecución de comandos.

Los eventos también puede ser usados para sincronizar la ejecución de comandos por distintos *streams*. En el código 2.6 se muestra un ejemplo en el que la

transferencia de información del *host* al dispositivo se hace por el *stream_1*, la ejecución del *kernel* por el *stream_2* y la transferencia de información del dispositivo al *host* por medio del *stream_3*. Esto requiere una sincronización entre *streams* por medio del uso de *eventos*. Esta sincronización hace que la ejecución de los *streams* se haga de forma secuencial. En las líneas 11-13 y 15-17 del código 2.6 se realiza la creación de *streams* y *eventos*, respectivamente. Tras la transferencia entre el *host* y el dispositivo por el *stream_1* en las líneas 35-38, se inserta el *evento_1* por el *stream_1* (línea 41 del código 2.6) con el fin de sincronizar el *stream_2* con el *stream_1* por medio del comando de la línea 44. Tras la ejecución del *kernel*, se realiza la misma operación de sincronización entre el *stream_2* y *stream_3* (líneas 47-56 del código 2.6). Al realizar la transferencia de información entre el dispositivo y el *host*, esta vez se hace la sincronización con el *host* (líneas 59-66 del código 2.6). Finalmente, se destruyen los *eventos* y *streams* que se han creado (líneas 73-80 del código 2.6).

```

1  int main()
2  {
3      int nElem = 2048;
4      float *h_A, *d_B, *d_C;
5      float *d_A, *d_B, *d_C;
6      int vectorSize = nElem * sizeof(float);
7      cudaStream_t stream_1, stream_2, stream_3;
8      cudaEvent_t evento_1, evento_2, evento_3;
9
10     /* Creacion de los streams */
11     cudaStreamCreate(&stream_1);
12     cudaStreamCreate(&stream_2);
13     cudaStreamCreate(&stream_3);
14     /* Creacion de los eventos */
15     cudaEventCreate(&evento_1);
16     cudaEventCreate(&evento_2);
17     cudaEventCreate(&evento_3);
18
19     /* Seleccion de GPU */
20     cudaSetDevice(0);
21
22     /* Reserva de memoria en el host */
23     h_A = (float *)malloc(vectorSize);
24     h_B = (float *)malloc(vectorSize);
25     h_C = (float *)malloc(vectorSize);
26     /* Reserva de memoria en el dispositivo */
27     cudaMalloc((void **)&d_A, vectorSize);
28     cudaMalloc((void **)&d_B, vectorSize);
29     cudaMalloc((void **)&d_C, vectorSize);
30
31     /* Inicializacion de vectores en el host */
32     inicializarVector(h_A, nElem);
33     inicializarVector(h_B, nElem);

```

```

34  /* Transferencia de informacion desde host al dispositivo en el
35  stream_1 */
36  cudaMemcpyAsync(d_A, h_A, vectorSize, cudaMemcpyHostToDevice,
37                stream_1);
38  cudaMemcpyAsync(d_B, h_B, vectorSize, cudaMemcpyHostToDevice,
39                stream_1);
40
41  /* Insercion del evento_1 en el stream_1 */
42  cudaEventRecord(evento_1, stream_1);
43
44  /* Sincronizacion del stream_2 con el stream_1 (evento_1) */
45  cudaStreamWaitEvent(stream_2, evento_1);
46
47  /* Lanzamiento del kernel */
48  dim3 nHilos(1024, 1, 1);
49  dim3 nBloques(nElem/nHilos.x, 1, 1);
50  vectorAddition<<<nBloques, nHilos, 0, stream_2>>>(d_A, d_B, d_C,
51                                                    nElem);
52
53  /* Insercion del evento_2 en el stream_2 */
54  cudaEventRecord(evento_2, stream_2);
55
56  /* Sincronizacion del stream_3 con el stream_2 (evento_2) */
57  cudaStreamWaitEvent(stream_3, evento_2);
58
59  /* Transferencia de resultado del dispositivo al host */
60  cudaMemcpyAsync(h_C, d_C, vectorSize, cudaMemcpyDeviceToHost,
61                stream_3);
62
63  /* Insercion del evento_3 en el stream_3 */
64  cudaEventRecord(evento_3, stream_3);
65
66  /* Sincronizacion del host con el evento 3 */
67  cudaEventSynchronize(evento_3);
68
69  /* Liberacion de memoria */
70  cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
71  free(h_A); free(h_B); free(h_C);
72
73  /* Destruccion de los eventos */
74  cudaEventDestroy(evento_1);
75  cudaEventDestroy(evento_2);
76  cudaEventDestroy(evento_3);
77
78  /* Destruccion de los streams */
79  cudaStreamDestroy(stream_1);
80  cudaStreamDestroy(stream_2);
81  cudaStreamDestroy(stream_3);
82 }

```

Código 2.6: Sincronización de streams mediante el uso de eventos CUDA.

La ejecución concurrente de diferentes comandos en las GPUs se puede realizar por medio de *streams*. Para ello las tareas que quieren ejecutarse de forma concurrente deben ser lanzadas por distintos *streams* de forma asíncrona. La disponibilidad de recursos de cómputo y buses de transferencia permitirán la ejecución concurrente de los comandos planificados. En el código 2.7 se muestra la ejecución de dos tareas de forma concurrente en la GPU. Para ello se han creado dos *streams* (líneas 6-7 del código 2.7). Los comandos de cada *kernel* se han lanzado en sus respectivos *streams*, en las líneas 10-14 y las líneas 17-21 del código 2.7 para las tareas 1 y 2, respectivamente. Con el fin de simplificar el código 2.7, los datos en el *host* para cada tarea se han definido con *h_1* y *h_2*, los datos en el dispositivo son *d_1* y *d_2*, los *streams* se representan con *stream_1* y *stream_2*, y finalmente, los *bytes*, *nBloques* y *nHilos* son los mismos para ambos *kernels*. En este caso, la secuencia de ejecución sería:

1. Transferencia desde el *host* al dispositivo de la tarea 1.
2. Coejecución de *kernel_1* junto con la transferencia desde el *host* al dispositivo de la tarea 2.
3. Si la transferencia de la tarea 2 termina antes de que termine la ejecución del *kernel_1*, y existen recursos de cómputo disponibles para la ejecución del *kernel_2*, se ejecutarían concurrentemente ambos *kernels*.
4. Al finalizar la ejecución del *kernel_1* puede ocurrir que el *kernel_2* siga ejecutándose, o que empiece su ejecución si no había recursos suficientes. En cualquier caso lo hará de forma concurrente con la transferencia de memoria desde el dispositivo al *host* de la tarea 1.
5. Tras la finalización de la transferencia de la tarea 1 y la ejecución del *kernel_2*, se ejecuta la transferencia desde el dispositivo al *host* por parte de la tarea 2.

```

1 int main()
2 {
3     cudaStream_t stream_1, stream_2;
4
5     /* Creacion de los streams */
6     cudaStreamCreate(&stream_1);
7     cudaStreamCreate(&stream_2);
8
9     /* Lanzamiento de la tarea 1 por el stream_1 */
10    cudaMemcpyAsync(d_1, h_1, bytes, cudaMemcpyHostToDevice,
11                   stream_1);
12    kernel_1<<<<nBloques, nHilos, 0, stream_1>>>>(d_1);

```

```

13  cudaMemcpyAsync(h_1, d_1, bytes, cudaMemcpyDeviceToHost,
14                stream_1);
15
16  /* Lanzamiento de la tarea 2 por el stream_2 */
17  cudaMemcpyAsync(d_2, h_2, bytes, cudaMemcpyHostToDevice,
18                stream_2);
19  kernel_2<<<<nBloques, nHilos, 0, stream_2>>>>(d_2);
20  cudaMemcpyAsync(h_2, d_2, bytes, cudaMemcpyDeviceToHost,
21                stream_2);
22
23  /* Destruccion de los streams */
24  cudaStreamDestroy(stream_1);
25  cudaStreamDestroy(stream_2);
26 }

```

Código 2.7: Ejecución de concurrente de comandos por medio del uso de *streams* en CUDA.

2.4. Planificador de Bloques y Warps

En esta sección se describen los mecanismos para la planificación de los bloques y *warps* de los *kernels* en ejecución en la GPU con el objetivo de comprender su funcionamiento en la arquitectura de NVIDIA. Este estudio nos permite caracterizar el comportamiento de los planificadores, lo que nos servirá de ayuda para la optimización de la ejecución concurrente de varias aplicaciones. Uno de los aspectos más importantes es el análisis de la interferencia en la planificación de bloques y *warps*, producida durante la coejecución en una GPU. El objetivo final de este estudio es mejorar el uso de los recursos y aumentar el rendimiento de la GPU.

La escalabilidad de las arquitecturas de unidades de procesamiento gráfico (GPUs) las ha convertido en alternativas eficaces a las CPUs para las cargas de trabajo paralelas. Las GPUs consiguen un rendimiento superior gracias al uso masivo de multihilos y al rápido cambio de contexto para ocultar las paradas en los *pipelines* y la latencia de acceso a la memoria. Los SMs de una GPU son similares a un procesador SIMD (*Single Instruction Multiple Data*), pero en este caso se denominan SIMT (*Single Instruction Multiple Threads*) ya que los hilos pueden seguir distintos caminos (*branches*). Estos procesadores disponen de varias unidades funcionales, cachés de datos e instrucciones, y unidades de obtención/decodificación de instrucciones.

Los bloques de hilos que ejecutan una *kernel* de una aplicación en una GPU también son denominados como CTAs (*Cooperative Thread Arrays*). Los hilos que

conforman un CTA son ejecutados en un mismo SM. Además, los hilos dentro de un bloque pueden cooperar a través de la memoria compartida y usar funciones de sincronización.

Los hilos de un bloque se agrupan a su vez en conjuntos denominados *warps*, normalmente compuestos por 32 hilos. Cada SM se encarga de crear, gestionar, planificar y ejecutar estos *warps*. Los hilos de un *warp* se ejecutan de forma conjunta, pero a partir de la arquitectura Volta cada uno de ellos posee su propio contador de programa y conjunto de registros por lo que pueden separarse y ejecutarse de forma independiente. En arquitecturas anteriores el procesador usa una máscara de hilos activos para indicar que hilos dentro de un *warp* ejecutan la instrucción actual.

Con este paradigma de multihilo masivo y múltiples registros, cada vez que un hilo (o un *warp*) se bloquea por una operación de alta latencia, las GPUs son capaces de realizar rápidamente un cambio de contexto para ocultar la latencia de ejecución con una penalización mínima. Por tanto, las GPUs modernas son capaces de lograr una mayor utilización media de los canales de acceso a memoria y un mayor rendimiento en comparación con los diseños tradicionales.

2.4.1. Planificador de bloques

El planificador de bloques es el encargado de gestionar los bloques de hilos para asignarlos a los diferentes SMs. El planificador de bloques sigue una política en la que intenta asignar el mayor número posible de bloques a cada SM (política *greedy*). El número máximo de bloques depende del uso de recursos de cada kernel y el límite superior lo marca la arquitectura.

En las GPUs de NVIDIA [58] al *hardware* encargado de planificar los bloques de hilos entre los distintos SMs se le denomina *GigaThread Engine*. NVIDIA no ha publicado información referente al funcionamiento de este planificador pero muchos investigadores han estudiado su comportamiento [33, 46, 53, 83]. Estos trabajos, entre otros, han analizado su mecanismo de actuación y propuesto nuevas técnicas para mejorar el rendimiento y la eficiencia de las GPUs. El comportamiento del planificador se corresponde con el de un planificador de tipo *round-robin* (RR). Los bloques asignados a los SMs se distribuyen, por defecto, siguiendo una planificación RR. Con esta distribución se intenta balancear la carga computacional de los SMs. Al inicio de la ejecución del kernel el primer bloque se lanza al primer SM, el segundo bloque al segundo SM y así sucesivamente hasta que se asignan todos los bloques o todos los SMs están llenos. El número máximo de bloques soportados por un SM depende del uso de recursos (hilos de

computo, número de registros, memoria compartida, etc.) por parte del kernel y de los límites hardware de la arquitectura. Una vez que un bloque finaliza, el planificador de bloques asigna otro bloque al SM hasta que todos los bloques han sido ejecutados. Por último, es interesante reseñar que los SMs están organizados jerárquicamente en *clusters* de procesamiento gráfico (GPC), de forma que el planificador tiene en cuenta esta jerarquía para la asignación de los bloques a los SMs.

2.4.2. Planificador de warps

Dentro de cada SM hay cuatro planificadores de *warps* (ver Tablas 2.4 y 2.5). Cada planificador de *warps* selecciona uno de entre todos los *warps* listos para ser ejecutados. Al igual que con los bloques, hay un número máximo de *warps* que pueden permanecer de forma residente en el procesador. Este número máximo depende de los requerimientos del *kernel* y de la capacidad de computación de la arquitectura. Más concretamente, para cada *warp* residente el SM tiene que mantener su contexto de ejecución (contador de programa, registros, etc) durante toda la vida del *warp*. Este número de *warps* residentes está relacionado con la utilización de la GPU, ya que para ocultar la latencia de las diferentes instrucciones hace falta tener *warps* residentes listos para ejecutarse.

2.5. Simulador de GPUs

GPGPU-Sim [12] es uno de los simuladores más usados para la propuesta de nuevas arquitecturas que permitan aplicar mejoras en las GPUs. Es un simulador de rendimiento de GPU a nivel de ciclo que se centra en la computación en GPU.

GPGPU-Sim simula programas que se componen de una parte de CPU y otra de GPU. Sin embargo, el modelo de microarquitectura de GPGPU-Sim da información de los ciclos en los que la GPU está ocupada, no modela la temporización de la CPU ni del bus de conexión PCIe o NVLink, es decir, no modela el tiempo de transferencia desde la memoria de la CPU a la GPU.

La intención de GPGPU-Sim es proporcionar un modelo para la investigación de arquitecturas más que modelar exactamente cualquier GPU comercial concreta. No obstante, GPGPU-Sim 3.1.0 obtiene una correlación del valor de IPC del 98,3% en una versión reducida del conjunto de *benchmarks* de Rodinia [18].

2.5.1. Arquitectura global de GPGPU-Sim

Una GPU modelada por GPGPU-Sim se compone de SIMT Cores (*Single Instruction Multiple Thread*) conectados a través de una red de conexión a particiones de memoria que interactúan con la memoria de la gráfica.

Un SIMT Core modela un procesador SIMD multihilo que equivale aproximadamente a lo que NVIDIA llama un *Streaming Multiprocessors* (SM) o lo que AMD llama una *Compute Unit* (CU). La organización de un SIMT Core se describe en la figura 2.13.

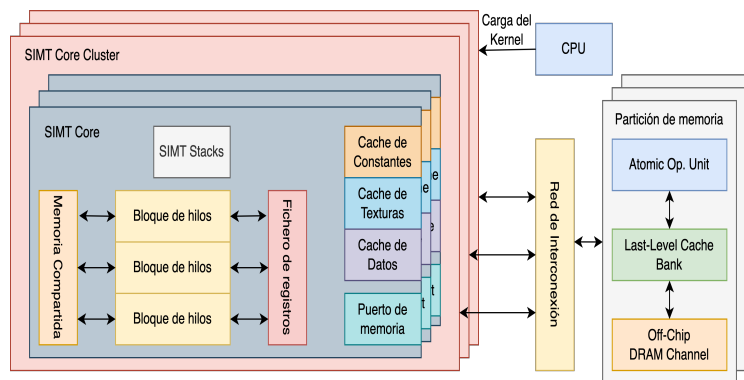


Figura 2.13: Arquitectura GPGPU-Sim.

GPGPU-Sim admite cuatro dominios de reloj independientes:

- El reloj del clúster del SIMT Core.
- El reloj de la red de interconexión.
- El reloj de la caché L2, que se aplica a toda la lógica de la unidad de partición de memoria excepto a la DRAM.
- El reloj de la DRAM.

Las frecuencias de reloj pueden tener cualquier valor arbitrario (no es necesario que sean múltiplos entre sí). En otras palabras, asumen la existencia de sincronizadores entre los dominios de reloj. En el modelo de simulación de la GPGPU-Sim, las unidades de los dominios de reloj adyacentes se comunican a través de búferes de cruce de reloj que se llenan a la frecuencia de reloj del dominio de origen y se vacían a la frecuencia de reloj del dominio de destino.

2.5.2. Componentes de GPGPU-Sim

En esta sección se describen de forma más detallada los distintos componentes que conforman la arquitectura de GPGPU-Sim.

SIMT Core Clusters

Los SIMT Cores se agrupan en SIMT Core Clusters. Los SIMT Cores de un SIMT Core Cluster comparten un puerto común a la red de interconexión, como se muestra en la figura 2.14. Cada SIMT Core Cluster tiene una única cola FIFO de respuesta que contiene los paquetes enviados desde la red de interconexión. Los paquetes se dirigen a la caché de instrucciones de un SIMT Core (si se trata de una respuesta de memoria que da servicio a un fallo de obtención de instrucciones) o a su canalización de memoria (unidad LDST). Los paquetes salen en el orden de entrada (FIFO). La cola FIFO de respuesta se bloquea si un core no puede aceptar el paquete en la cabecera de la cola FIFO. Para generar solicitudes de memoria en la unidad LDST, cada SIMT Core tiene su propio puerto de inyección en la red de interconexión. Sin embargo, el búfer del puerto de inyección es compartido por todos los SIMT Cores de un clúster.

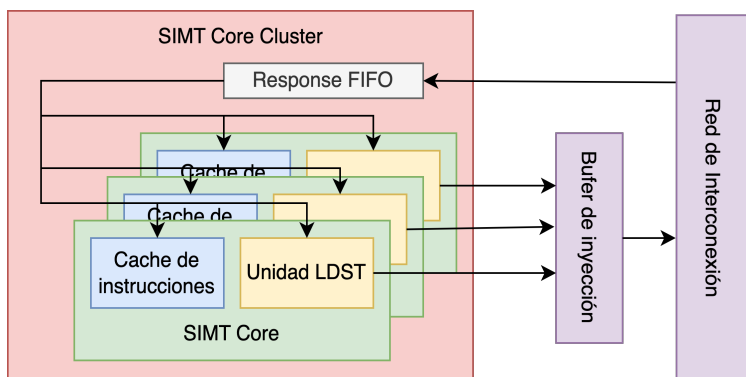


Figura 2.14: SIMT Core Cluster en GPGPU-Sim.

Red de interconexión

La red de interconexión se encarga de las comunicaciones entre los *SIMT Core Clusters* y las particiones de memoria. Para simular la red de interconexión interconectan el simulador booksim [35] con GPGPU-Sim. Booksim es capaz de

simular redes Tori [4] y Fly [8] basadas en canales virtuales y es altamente configurable.

En GPGPU-Sim modifican booksim para obtener su propia red de interconexión (Intersim). Intersim tiene su propio dominio de reloj. El booksim original sólo admite una única red de interconexión pero en GPGPU-Sim realizan algunos cambios para poder simular dos redes de interconexión: una para el tráfico desde los *SIMT Core Clusters* a las particiones de memoria y otra red para el tráfico desde las particiones de memoria de vuelta a los *SIMT Core Clusters*. Esta es una forma de evitar las dependencias circulares que podrían causar bloqueos de la red de interconexión.

Los *SIMT Core Clusters* no se comunican directamente entre sí y, por tanto, no existe la noción de tráfico de coherencia en la red de interconexión. En GPGPU-Sim sólo hay cuatro tipos de paquetes: (1) read-request y (2) write-requests enviados desde los *SIMT Core Clusters* a las particiones de memoria y (3) read-replies y (4) write-acknowledges enviados desde las particiones de memoria a los *SIMT Core Clusters*.

Partición de memoria

El sistema de memoria en GPGPU-Sim se modela mediante un conjunto de particiones de memoria. Como se muestra en la figura 2.15, cada partición de memoria contiene un banco de caché L2, un planificador de acceso a la DRAM y el canal de DRAM fuera del chip. La ejecución funcional de las operaciones atómicas también se produce en las particiones de memoria en la fase de ejecución de operaciones atómicas. A cada partición de memoria se le asigna un subconjunto de direcciones de memoria física del que es responsable. Por defecto, el espacio de direcciones lineal global se intercala entre las particiones en trozos de 256 bytes. Esta partición del espacio de direcciones junto con el mapeo detallado del espacio de direcciones a la fila, bancos y columnas de la DRAM en cada partición es configurable.

La caché L2 (cuando está activada) atiende las solicitudes de memoria de textura entrantes y las que no son de textura. Cuando hay un fallo en la caché, el banco de caché L2 genera solicitudes de memoria al canal de DRAM para que sean atendidas por la DRAM fuera del chip. El simulador permite usar distintos tipos de memoria, GDDR y HBM entre otras.

mejor rendimiento en términos de *IPC* global que cuando los *kernels* se ejecutan en solitario.

$$STP = \sum_{k=1}^K \frac{IPC_k^{shared}}{IPC_k^{alone}} \quad (2.2)$$

- **ANTT** (*Average Normalized Turnaround Time*). Esta métrica representa el retardo relativo para un conjunto de *kernels* (K). La ecuación 2.3 es usada para calcular el *ANTT*. Cuanto menor es el valor de esta métrica, mejor es el resultado obtenido.

$$ANTT = \frac{1}{K} \left(\sum_{k=1}^K \frac{1}{NP_k} \right) \quad (2.3)$$

- **Fairness**. Esta métrica representa la equidad durante la ejecución de un par de *kernels* i y j . El *fairness* viene dado por la ecuación 2.4. Esta métrica oscila entre 0 y 1, donde 1 indica que el reparto ha sido lo más justo posible.

$$Fairness = \min_{i,j} \left(\frac{NP_i}{NP_j} \right) \quad (2.4)$$



UNIVERSIDAD
DE MÁLAGA

3 Ejecución solapada de transferencias con kernels

En este capítulo se describen los elementos necesarios y los métodos aplicados para la ejecución concurrente de tareas o aplicaciones en una GPU. Además, se estudia la planificación dinámica de estas tareas para proponer una heurística que ayude a encontrar el mejor orden de ejecución para dicho conjunto de tareas.

Una aplicación o tarea está compuesta por una serie de comandos que deben ser ejecutados para su finalización. Las tareas están compuestas por tres tipos de comandos básicos: transferencias de datos desde la CPU o *host* a la GPU o dispositivo (*HtD*), ejecución del *kernel* (*K*) y transferencia de los datos resultantes de la ejecución del *kernel* desde el dispositivo al *host* (*DtH*). Normalmente, las tareas están compuestas por uno o más *kernels*, mientras que las transferencias pueden ser opcionales dependiendo de las necesidades de cada tarea. En la sección 3.1 se hace un repaso del estado del arte de los modelos de ejecución concurrente de tareas en GPUs, mostrando los trabajos más destacados en el análisis de las transferencias entre el *host* y los dispositivos, la ejecución de los *kernels* y la ejecución concurrente de los comandos que conforman múltiples tareas.

Los comandos lanzados en la misma cola de comandos (*stream* en la terminología CUDA) son ejecutados en orden de llegada, por lo que no puede existir concurrencia de comandos lanzados en un mismo *stream*. La concurrencia entre tareas se realiza mediante la ejecución de comandos asíncronos en *streams* diferentes. En la sección 3.2 se describe como los comandos de transferencia o ejecución de *kernels* deben ser lanzados de forma asíncrona por distintos *streams* para su coejecución.

Una coejecución óptima de tareas requiere una buena estimación de los tiempos de ejecución de cada comando y así poder hacer una buena planificación. En

la sección 3.3 se detalla un modelo simple de ejecución de *kernels* que es ampliamente usado en la literatura, y un modelo de transferencia al que le hemos añadido una serie de modificaciones que nos permiten modelar las transferencias de datos de forma más exacta.

Una serie de conceptos generales sobre planificación de tareas se detallan en la sección 3.4. Las heurísticas de planificación se describen en la sección 3.5 junto a la heurística propuesta en esta tesis. En la sección 3.6 se detallan los experimentos realizados para validar y analizar las aportaciones realizadas en este campo.

3.1. Estado del Arte

El análisis del comportamiento de una GPU requiere de modelos que permitan predecir su rendimiento. Estos modelos a su vez pueden aplicarse a sistemas heterogéneos para mejorar su prestaciones. Los apartados siguientes describen los trabajos más relevantes en el análisis de las transferencias de memoria entre el *host* y los *dispositivos* en ambas direcciones, la ejecución de los *kernels* y la coejecución de comandos.

3.1.1. Modelado de Transferencias

Una tarea diseñada para ser ejecutada en un sistema heterogéneo, con CPU y una GPU discreta, requiere de una, ninguna o varias transferencias de datos desde el *host* al dispositivo y viceversa. Estas transferencias pueden consumir grandes cantidades de tiempo en la ejecución global de una tarea y pueden afectar en el rendimiento de la política de planificación aplicada con la que se decide el orden de ejecución de las tareas lanzadas. El impacto de las transferencias en el tiempo total de ejecución viene dado sobre todo por el hecho de que la ejecución del *kernel* debe esperar a la finalización de la transferencia desde el *host* al dispositivo. Además, en las aplicaciones diseñadas para sistemas heterogéneos, se debe tener en cuenta el tipo de bus de conexión de datos utilizado.

La reserva de memoria en el *host* se hace con memoria paginable por defecto. Desgraciadamente, la GPU no puede acceder a los datos alojados en la memoria paginable del *host* de forma directa, por lo que cuando se invoca una transferencia de datos desde la memoria del *host* paginable a la memoria del dispositivo, el controlador de CUDA asigna espacio temporalmente en la memoria *pinned* del *host*, copia esos datos a dicha memoria reservada temporalmente y, finalmente, transfiere los datos a la memoria de la GPU. Como se puede apreciar en el lado

izquierdo de la figura 3.1, la memoria no paginable o *pinned* se utiliza como zona de paso para las transferencias desde el *host* al dispositivo o viceversa. El coste que conlleva esta transferencia se puede evitar reservando la memoria necesaria en el *host* en la memoria *pinned*. En el lado derecho de la figura 3.1 se muestra el recorrido de los datos cuando se hace una transferencia desde el *host* al dispositivo utilizando memoria *pinned*, por lo que se ahorra la transferencia desde la memoria paginable a la memoria *pinned*, reduciendo el tiempo necesario para la transferencia de datos.

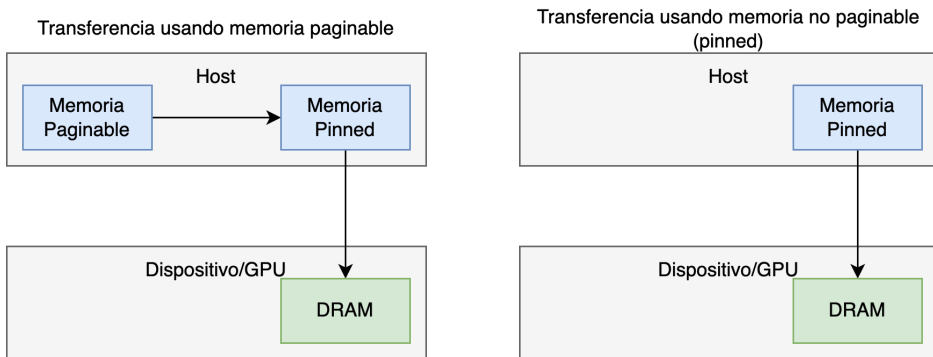


Figura 3.1: Izquierda: transferencia de datos usando memoria paginable. El *host* requiere reservar memoria *pinned* para transferir los datos a la memoria del dispositivo. Derecha: Transferencia de datos usando memoria no paginable (*pinned*). El *host* utiliza la memoria *pinned* para comunicarse de forma directa con el dispositivo.

El uso de distintos tipos de memoria para la transferencia de datos hace que se planteen distintos enfoques a la hora de modelar las transferencias en los sistemas heterogéneos. En [14] se presenta un modelo del rendimiento de aceleradores como las GPUs. En la transferencia de datos, usan el número de datos a transmitir por un grupo de tareas y usando un modelo lineal de rendimiento del bus de datos PCIe predicen el tiempo necesario para cada transferencia de datos. El trabajo [26] presenta también un modelo lineal para conocer el tiempo requerido en transferencia de datos para *kernels* como la multiplicación de matrices. En [94] se usa la arquitectura del dispositivo, la dirección a la que se va a hacer la transferencia y su tipo para calcular el tiempo de transferencia de forma más precisa. El modelo presentado por este estudio se denomina LogGP [5, 22]. Las GPUs utilizadas en este estudio son clasificadas en tres arquitecturas distintas según el número de motores DMA presentes en su arquitectura: (1) GPUs con un único DMA en las que se usa la sincronización implícita (GTX 480 y GTX 680),

(2) GPUs con un único DMA donde no se usa la sincronización implícita (GTX Titan), y (3) GPUs con dos DMAs sin sincronización implícita (Tesla K20c, S2050 y GTX980). Además, modelan cada arquitectura para conocer el nivel de solapamiento que se puede conseguir entre transferencias y computación. El estudio [88] mejora el modelo presentado por [94] para transferencias de memoria con zonas de memoria no alineadas.

El uso de memoria unificada [74] es otro mecanismo para transferencias de memoria entre el *host* y el dispositivo. Este tipo de memoria simplifica la programación de las aplicaciones heterogéneas sin la necesidad de copiar manualmente los datos del *host* al dispositivo, facilitando la introducción de soporte para la aceleración en GPU en un amplio catálogo de lenguajes de programación. Como contrapartida se pierde el control sobre las transferencias, ya que el *runtime* es el encargado de su gestión. En UMH [110] se analizan las ventajas respecto a la programación y el rendimiento en el ancho de banda al usar de la memoria unificada.

En esta tesis se presenta un modelado de transferencias que, a diferencia con el modelo propuesto en [94], tiene en cuenta la posibilidad de transferencias simultáneas en sentidos opuestos y con la posibilidad de cambiar la planificación durante la ejecución de un conjunto de tareas.

3.1.2. Modelado del tiempo de ejecución de Kernels

En el estado del arte existe un amplio catálogo de trabajos que estudian la caracterización del rendimiento de *kernels* en GPUs. En [55] proponen GROPHECY en el que estudian el rendimiento de los *kernels* para optimizar su comportamiento en las GPUs. Sim J. et al. [86] estudian los cuellos de botella producidos por los *kernels* en la arquitectura Turing. Konstantinidi E. et al. [43] proponen un modelo predictor del tiempo de ejecución de aplicaciones CUDA. En este trabajo usan un modelo que permite estimar las limitaciones del *kernel* por medio de una serie de métricas. En entornos OpenCL, donde se pueden aplicar modelos usados en arquitecturas CUDA, Lemeire J. et al. [47] usan métricas como el número de operaciones y bytes por segundo, el porcentaje de ocupación de los recursos de la GPU, el número de ciclos por instrucción o la latencia para obtener una gráfica de rendimiento de los *kernels*. En [39] identifican una serie de cuellos de botella como la latencia producida por la memoria, la divergencia en los hilos de ejecución dentro de un mismo bloque o *warp*, utilización de recursos y el paralelismo. En él proponen un modelo de rendimiento que les permite conocer el cuello de botella y su causa por medio del uso de un *profiler*. El uso de esta técnica les

permite implementar *kernels* tanto en OpenCL como con CUDA. Liu B. et al. [52] usan un modelo lineal por *software* que les permite predecir el tiempo de ejecución de los *kernels* en arquitecturas NVIDIA y AMD, siendo un modelo simple y fácilmente aplicable.

3.1.3. Ejecución Concurrente de Comandos

La colas de comandos *software*, *streams* en terminología CUDA [74], permiten la ejecución concurrente de comandos en las GPUs de NVIDIA. Estos *streams* permiten la ejecución de comandos que se encuentran en colas distintas, siempre que haya recursos suficientes para poder ejecutarse simultáneamente. El soporte *hardware* que permite la ejecución concurrente de comandos en arquitecturas NVIDIA se denomina Hyper-Q.

Gomez-Luna J. et al. [28] estudian el solapamiento de comandos de transferencia de datos y computación con el uso de *streams*. En dicho trabajo se centran en encontrar, en tiempo de ejecución, el número óptimo de *streams* necesarios para mejorar el rendimiento de los comandos en una GPU. En [52] muestran como el particionado de los datos y su planificación afecta al rendimiento de la ejecución de un *kernel* en un acelerador. En este estudio no se contempla la transferencia de datos en sentidos opuestos de forma concurrente, las cuales se pueden realizar en aceleradores que poseen dos motores DMA. Li Z. et al. [49] estudian la ejecución concurrente de comandos en plataformas heterogéneas MIC usando hStreams [87], una plataforma para la ejecución de aplicaciones en plataformas heterogéneas compuestas por una CPU y GPUs. Basaran C. et al. [13] estudian el uso de planificadores apropiativos para la ejecución de comandos de forma concurrente en las GPUs.

En este capítulo presentamos un modelo para ejecutar concurrentemente un conjunto de tareas de forma eficiente en GPUs de NVIDIA. El modelo presentado mejora los trabajos anteriores mediante el solapamiento de comandos de transferencia de datos y computación de forma más precisa. Con esta propuesta se pueden manejar casos más complejos y realistas, donde los distintos hilos de la CPU lanzan tareas de forma simultánea a la GPU.

3.2. Lanzamiento Asíncrono de Comandos

El lanzamiento de comandos desde el *host* hacia la GPU se lleva a cabo por medio de una cola *software*. Los comandos suelen clasificarse en dos tipos: coman-

dos de transferencia de datos entre el *host* y el dispositivo o GPU; y comandos de ejecución de *kernels* en la GPU. La ejecución de comandos por una misma cola *software* es equivalente a una ejecución síncrona, ya que el inicio de la ejecución de un comando depende de la finalización del comando previo, por lo que el solapamiento entre comandos lanzados por una misma cola *software* no es posible. Por otro lado, la ejecución asíncrona puede llevarse a cabo por medio del uso de distintas colas *software*. De esta forma los comandos lanzados por distintas colas *software* no son dependientes entre sí y pueden solapar su ejecución siempre que existan suficientes recursos *hardware* donde poder ejecutarse. El tipo y cantidad de comandos que pueden ejecutarse concurrentemente depende de las características *hardware* del dispositivo. El solapamiento de comandos de transferencia de datos depende del número de motores DMA que posea el dispositivo, ya que para un solapamiento de comandos de transferencia de datos entre el dispositivo y el *host* son necesarios dos DMAs en la GPU. Más concretamente, los dos motores DMA permiten solapar un comando *HtD* y otro comando *DtH*.

Los aceleradores pueden ser clasificados en distintos tipos [94] atendiendo al número de motores DMA y del mecanismo de sincronización de comandos que posean (con sincronización implícita o sin ella). La sincronización implícita se produce cuando la ejecución de un comando dentro de un *stream* requiere de la finalización del comando previo lanzado por ese mismo *stream* [74]. Además, los *kernels* lanzados por distintas colas *software* no podrán ejecutarse concurrentemente si alguno de los *kernels* ocupa todos los recursos *hardware* del dispositivo. Por tanto, el solapamiento de comandos dependerá del número de motores DMA, de los recursos de cómputo disponibles y de la planificación de los comandos. Las colas *software* de comandos poseen un mecanismo para marcar dependencias entre distintas colas de comandos mediante eventos, de forma que un hilo en el *host* puede comprobar el estado de un evento con el fin de saber si un comando ha comenzado o finalizado su ejecución.

En esta tesis nos hemos centrado en la arquitectura propuesta por NVIDIA con las GPUs K20c, GTX980 y Titan X. Las características de dichas GPUs se plasman en la tabla 3.1.

En la figura 3.2 se muestra la ejecución de cuatro tareas en una GPU con dos motores DMA, por lo que se permite el solapamiento de comandos de transferencia de datos en direcciones opuestas. En este caso, vamos a suponer que los *kernels* consumen todos los recursos de cómputo de la GPU, por lo que no existe ejecución concurrente de *kernels*. La guía de programación de CUDA [74] recomienda lanzar los comandos de una misma tarea por un mismo *stream*. El hilo del *host* agrupa los comandos de una misma tarea y los lanza por un mismo *stream*, esto permite que los dos motores DMA puedan funcionar al mismo tiempo. Los

GPU	Motores DMA	Sincronización	Arquitectura	PCIe
K20c	2	Dentro del mismo <i>stream</i>	Kepler	2.0
GTX980	2	Dentro del mismo <i>stream</i>	Maxwell	2.0
Titan X	2	Dentro del mismo <i>stream</i>	Pascal	3.0

Tabla 3.1: GPUs K20c, GTX980 y Titan X de NVIDIA. Número de motores DMA, tipo de sincronización y donde se produce, arquitectura de las GPUs usadas en esta tesis y bus de conexión PCIe utilizado.

comandos de cada tarea se indican por HtD_s (transferencia *host to device*), K_s (ejecución de *kernel*) y DtH_s (transferencia *device to host*), donde s indica el identificador del *stream* en el que se ejecutan. La sincronización impuesta por el *stream* a los comandos lanzados por el mismo mantiene las dependencias internas de dicha tarea. En el ejemplo, se puede observar que el *stream 0* comienza la ejecución de su tarea con la transferencia HtD_0 , tras ello comienza la ejecución de K_0 junto a HtD_1 , gracias a que son comandos lanzados por distintos *streams*. La ejecución concurrente de los comandos HtD_2 , K_1 y DtH_0 se debe a que además de lanzarse los comandos por distintos *streams*, existen dos motores DMA que permiten solapar las transferencias en sentidos opuestos junto a la ejecución del *kernel* (K_1) en los núcleos de la GPU.

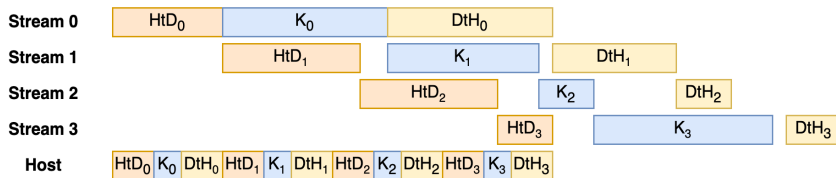


Figura 3.2: Esquema de lanzamiento en GPUs NVIDIA con dos motores DMA. El *host* agrupa los comandos por tareas y los lanza en *streams* diferentes. La dependencia interna entre los comandos de una tarea es mantenida por medio de la sincronización implícita de cada *stream*.

3.3. Estimación del Tiempo de Ejecución de Comandos

La aplicación de heurísticas para la planificación de un conjunto de tareas puede mejorar su rendimiento realizando una estimación previa del tiempo ne-

cesario para ejecutar los comandos a planificar. En esta sección se describe la estimación del tiempo necesario para realizar transferencias de datos entre el *host* y el dispositivo, que depende del tipo de memoria reservada y de la existencia de transferencias en sentido opuesto, y también la estimación del tiempo de ejecución de los *kernels*.

3.3.1. Transferencias de Memoria

Las aplicaciones desarrolladas para su ejecución en la GPU, requiere de una reserva de memoria previa antes del envío de datos desde el *host* al dispositivo. La guía de programación de CUDA [74] recomienda el uso de memoria *pinned* para explotar al máximo el ancho de banda del bus de comunicación. La memoria *pinned* se aloja en la memoria DRAM del *host* y no se puede trasladar al disco. Las transferencias de memoria con el dispositivo o GPU requiere del uso de memoria *pinned*. Si se usa memoria paginable, cuando se hace una transferencia desde el *host* a la GPU se hace una copia de memoria desde la memoria paginable a la *pinned* y posteriormente a la GPU (lado izquierdo de la figura 3.1). El mismo recorrido pero en sentido inverso se recorrería cuando la transferencia de datos se produce entre la GPU y el *host*. En el caso del uso de memoria *pinned*, la transferencia entre el *host* y el dispositivo se hace de forma directa por medio del DMA (lado derecho de la figura 3.1). El sistema operativo limita la cantidad de memoria *pinned* que se puede reservar, y si se hace un uso elevado de este recurso podría degradar el rendimiento global de la memoria. En [54] indican que el uso de memoria *pinned* mejora la velocidad de transferencia, pero puede consumir demasiado tiempo en la reserva de memoria, así que es necesario llegar a un compromiso entre el tiempo de transferencia y la reserva de memoria.

De acuerdo a nuestros experimentos, la reserva de memoria *pinned* durante la transferencia de memoria paginable causa una bajada en el uso del ancho de banda de la memoria respecto a una transferencia desde memoria *pinned*. Además, la transferencia de datos concurrente en sentido contrario puede afectar al ancho de banda. Hemos realizado una serie de experimentos para ver el comportamiento de los distintos comandos de transferencia. En primer lugar, hemos ejecutado transferencias *HtD* y *DtH* en solitario para transferencias de datos desde memoria *pinned* y paginable usando distintos tamaños de datos para calcular el ancho de banda. En segundo lugar, hemos lanzado dos comandos de transferencia de datos del mismo tipo (paginable o *pinned*), simultáneamente, pero en direcciones opuestas, para estudiar el impacto en el ancho de banda de las transferencias en ambos sentidos. En tercer lugar, hemos hecho un experimento similar pero usando una transferencia desde memoria *pinned* en una dirección y una transferencia

usando memoria paginable en sentido contrario. Finalmente, hemos intentado solapar dos comandos de transferencia en el mismo sentido.

Los experimentos se han realizado usando CUDA 10 en tres GPUs de NVIDIA pertenecientes a arquitecturas diferentes cuyas características se plasman en la tabla 3.1. Las transferencias de datos se han realizado con tamaños que van desde 1 MB a 512 MB. Cada experimento ha sido ejecutado cincuenta veces para obtener un ancho de banda medio.

En la figura 3.3 se muestran los resultados del primer y segundo experimento para cada GPU. Respecto al ancho de banda correspondiente a las transferencias de memoria en solitario en cualquier dirección usando memoria paginable y *pinned*, se aprecia que el ancho de banda usando memoria *pinned*, llamados *Pin. HtD* (rojo oscuro) y *Pin. DtH* (verde oscuro), es mayor que el ancho de banda conseguido por memoria paginable, llamados *Pag. HtD* (azul oscuro) y *Pag. DtH* (azul). Además, el ancho de banda para memoria *pinned* es estable para todos los tamaños de transferencia de datos, mientras que en transferencias de datos para memoria paginable, para tamaños más pequeños es variable. Este comportamiento se debe al *runtime*, ya que para transferencias de datos con un tamaño elevado se produce una fragmentación de los datos a transmitir, lo que produce una pérdida en el ancho de banda alcanzado [108]. También hemos realizado experimentos donde dos transferencias de datos desde memoria *pinned* *HtD* y *DtH* han sido lanzadas concurrentemente. Así, la línea roja (*Pin. HtD* - *Pin. DtH*) muestra el ancho de banda obtenido cuando una transferencia *HtD* se solapa con una transferencia *DtH* con memoria *pinned*. De forma similar, la línea verde (*Pin. DtH* - *Pin. HtD*) muestra el resultado cuando una transferencia *DtH* se ejecuta concurrentemente con una transferencia *HtD* en memoria *pinned*. Esto muestra que el solapamiento de transferencias de memoria reduce el ancho de banda en cualquiera de las tres GPUs. Hemos realizado experimentos similares con memoria paginable y hemos confirmado que las transferencias concurrentes con memoria paginable no son posibles. Ésto se debe a que el *host* no puede hacer transferencias simultáneas entre la memoria paginable y la memoria *pinned* controlada por el *runtime* de CUDA.

La figura 3.4 ilustra los resultados del tercer experimento para cada GPU. De esta forma, un par de transferencias se ejecutan al mismo tiempo pero en sentido contrario donde una transferencia se hace usando memoria paginable y la otra con memoria *pinned*. La línea azul oscura (*Pag. HtD* - *Pin. DtH*) muestra el ancho de banda obtenido cuando una transferencia *HtD* usando memoria paginable se solapa con una transferencia *DtH* usando memoria *pinned*. El resto de combinaciones posibles también se muestran en la figura. Una vez más las transferencias realizadas con memoria paginable alcanzan un ancho de banda inferior

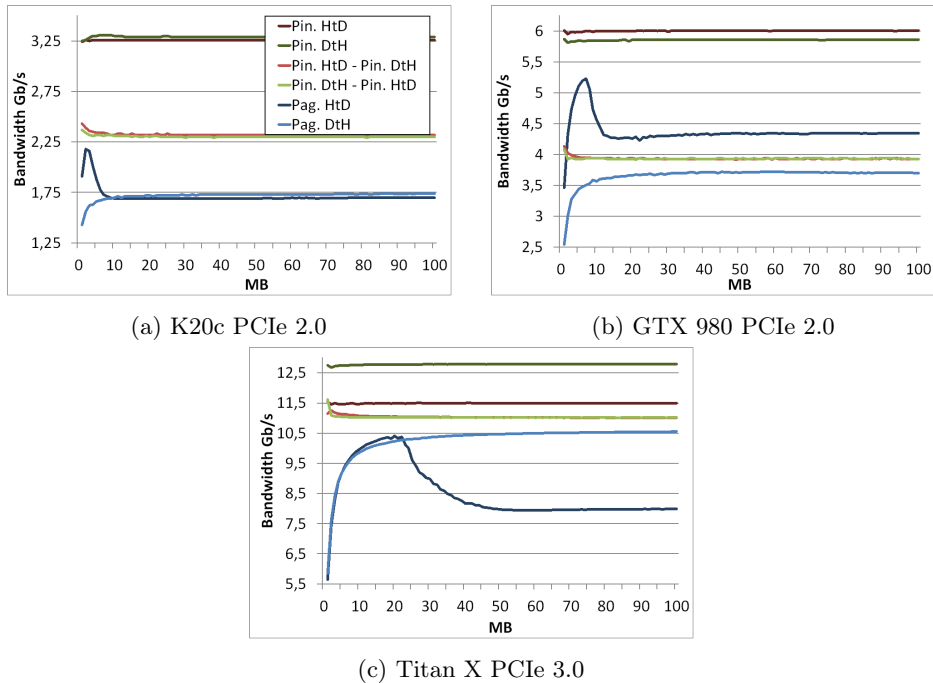


Figura 3.3: Ancho de banda medio obtenido para las GPUs K20c, GTX 980 y Titan X usando transferencias de memoria *pinned* y paginable. Los resultados muestran el rendimiento obtenido cuando se ejecutan solas o se solapan con otra transferencia usando la misma memoria pero en sentido opuesto.

a las transferencias con memoria *pinned*, excepto para la GPU GTX 980 donde el ancho de banda es similar cuando el tamaño de las transferencias superan los 10 MB, aunque las transferencias con memoria *pinned* siguen teniendo un ancho de memoria mayor a las transferencias de memoria paginable. Este experimento muestra que el ancho de banda de la transferencia usando memoria paginable se ve menos influenciado cuando se solapa con una transferencia de memoria *pinned*, mientras que la transferencia de memoria *pinned* reduce su ancho de banda de forma considerable cuando se solapa con una *paginable*, como se ilustra en la figura 3.3. Finalmente, en el cuarto experimento, hemos confirmado que las dos transferencias en el mismo sentido no pueden realizarse ni para memoria *pinned*, ni para memoria paginable.

El tiempo necesario para realizar la transferencia de datos con comandos *HtD* y *DtH* se puede estimar modelando el bus de conexión. Werkhoven et al. [94] pro-

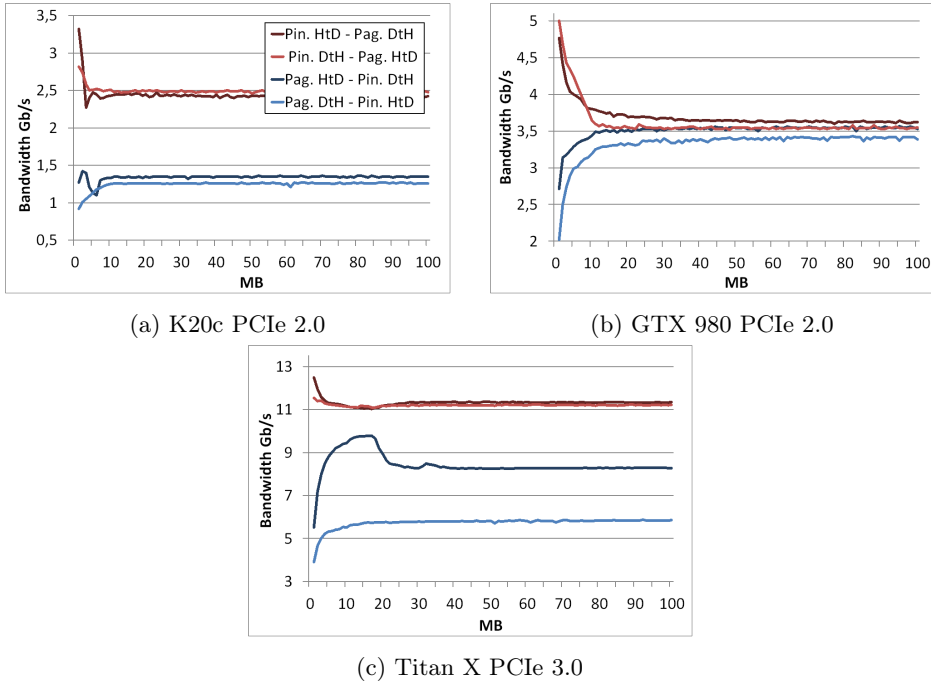


Figura 3.4: Ancho de banda medio obtenido para las GPUs K20c, GTX 980 y Titan X con diferentes combinaciones de transferencias con memoria paginable y *pinned* en direcciones opuestas.

ponen un modelo para el bus de conexión *PCI Express* denominado LogGP [5, 22]. Los parámetros usados en este estudio pueden medirse mediante una aplicación de *benchmark* de forma simple. Este modelo también tiene en cuenta la transferencia de datos de forma simultánea en sentidos opuestos. El modelo es correcto cuando las transferencias en sentidos opuestos se solapan completamente, pero no es preciso cuando el solapamiento se produce de forma parcial. Lázaro et al. [44] modifican LogGP para obtener un modelo más preciso que permite estimar correctamente tiempos de transferencia con solapamiento parcial. No obstante, este nuevo modelo sólo considera transferencias de datos desde memoria *pinned*, por lo que no sería válido para aplicaciones GPU que usan transferencias de memoria paginable. En nuestro trabajo proponemos un modelo más genérico donde se consideran ambos tipos de transferencia de datos, usando memoria *pinned* y

paginable. Este nuevo modelo se puede expresar mediante la siguiente ecuación:

$$t_t = L + o + (1 - \lambda)kG^1 + (1 - \alpha)\lambda kG^2 + \alpha\lambda kG^3 \quad (3.1)$$

En la ecuación 3.1, L es el límite superior de la latencia incurrida cuando se envía un mensaje de k bytes desde un punto a otro. El parámetro o es el tiempo en el que la CPU está involucrada en el registro de la petición DMA con el controlador. El factor de ponderación α es un parámetro que toma valor 0 cuando las dos transferencias desde memoria *pinned* se ejecutan concurrentemente en direcciones opuestas y toma valor 1 cuando una de las transferencias desde memoria *pinned* se ejecuta al mismo tiempo que una transferencia desde memoria paginable. Además, G^1 es la inversa del ancho de banda cuando las transferencias se realizan solas, G^2 es la inversa del ancho de banda cuando dos transferencias desde memoria *pinned* se hacen concurrentemente y G^3 es la inversa del ancho de banda cuando una transferencia desde memoria *pinned* se hace concurrentemente con una transferencia desde memoria paginable. Por último, λ es la fracción de bytes transferidos concurrentemente. La tabla 3.2 muestra los valores medios obtenidos para estos parámetros en las tres GPUs de NVIDIA usadas en esta tesis. En el caso de las transferencias desde memoria paginable, no hay G^2 porque no se pueden ejecutar dos transferencias de memoria paginable al mismo tiempo. Además, G^1 y G^3 para transferencias desde memoria paginable y G^3 para transferencias de memoria *pinned*, corresponden a valores donde el ancho de banda es estable. Para transferencia de memoria cortas el ancho de banda es obtenido usando una función de PCIe. Los valores de $L + o$ se muestran en las columnas *LoHtD* y *LoDtH*. La velocidad máxima del PCIe para cada GPU se muestra en la columna *PCIe*.

	Paginable				Pinned					PCIe	
	LoHtD (ms)	LoDtH (ms)	G^1 (s/Gb)	G^3 (s/Gb)	LoHtD (ms)	LoDtH (ms)	G^1 (s/Gb)	G^2 (s/Gb)	G^3 (s/Gb)		
K20c	<i>HtD</i>	0,0135	0,0193	1,70	1,34	0,0177	0,0155	3,26	2,32	2,41	2,0 x16 5Gbit/s
	<i>DtH</i>			1,74	1,26			3,29	2,30	2,48	
GTX 980	<i>HtD</i>	0,0113	0,0141	4,35	3,54	0,0134	0,0121	6,01	3,93	3,63	2,0 x16 5Gbit/s
	<i>DtH</i>			3,68	3,39			5,86	3,94	3,54	
TITAN X	<i>HtD</i>	0,0081	0,0098	8,00	8,28	0,0096	0,0089	11,49	11,02	11,33	3,0 x16 8Gbit/s
	<i>DtH</i>			10,57	5,81			12,79	11,02	11,2	

Tabla 3.2: Parámetros del modelo de transferencia para las GPUs K20c, GTX 980 y Titan X para distintos tipos de transferencias de memoria.

Este modelo de predicción del tiempo empleado en completar las transferencias generaliza los modelos propuestos previamente [94, 44]. Para su aplicación no

es necesario calcular de antemano λ y α para cada comando de transferencia, solo los parámetros L , o , G^1 , G^2 y G^3 del modelo para cada arquitectura. Cuando un comando de transferencia (HtD o DtH) es seleccionado, la estimación del tiempo es calculada usando un modelo no solapado. Si otro comando de transferencia en sentido opuesto (DtH o HtD) está listo para su lanzamiento, entonces el tiempo de ejecución de ambos comandos es recalculado asumiendo un solapamiento total y el tiempo mínimo de ambos tiempos es seleccionado. Este tiempo refleja la cantidad de solapamiento entre ambas transferencias y puede usarse para calcular el valor de λ para comandos de transferencia de datos grandes. Finalmente, el tiempo predicho para este comando puede ser actualizado utilizando la ecuación 3.1.

3.3.2. Ejecución de Kernels

En el estado del arte existen muchos trabajos que pretenden caracterizar el rendimiento de los *kernels* al ser ejecutados en los aceleradores. Por ejemplo, en OmpSs [24] y StarPU [10] obtienen el tiempo de ejecución de los *kernels* mediante ejecuciones iniciales de los mismos. Otros trabajos presentan modelos complejos que buscan aplicar técnicas de optimización para mejorar su rendimiento, pero nuestro objetivo es únicamente predecir el tiempo de ejecución de los *kernels*. En esta tesis se emplea un modelo lineal simple planteado en [44] el cual es similar al usado por Liu et al. [52]. En él, el tiempo de computación estimado viene dado por la ecuación 3.2, donde m es el número de elementos de datos de entrada, η es la tasa de computación definida a partir del tiempo de ejecución para cada dato m y γ es la latencia de invocación del kernel. En nuestro modelo de ejecución de comandos concurrentes solo necesitamos mantener un registro de los parámetros η y γ que se obtienen de ejecuciones previas de los *kernels* a planificar.

$$T = \eta \cdot m + \gamma \quad (3.2)$$

3.4. Teoría de la Planificación

En esta sección se describe como planificar un conjunto de tareas, durante la ejecución de las mismas, con el fin de reducir el tiempo necesario para ejecutar dichas tareas. La planificación dinámica de tareas es una labor importante en sistemas donde la carga de trabajo es heterogénea, ya que en estos sistemas resulta fundamental aplicar técnicas de balanceo de carga para mejorar el rendimiento. El planificador de tareas de la GPU no siempre es capaz de seleccionar el mejor

orden de lanzamiento. De hecho, el planificador de las GPUs de NVIDIA, llamado HyperQ, puede alterar el orden de ejecución y empeorar el comportamiento durante la ejecución. Esto es debido a que no tiene en cuenta las características de las tareas que se encuentran en la colas *hardware* para mejorar el solapamiento entre ellas. La obtención de un buen orden de ejecución que permita maximizar el solapamiento entre tareas de cómputo y transferencia de datos depende del análisis de las tareas que se quieren coejecutar.

El problema de la planificación de tareas en una GPU se puede estudiar mediante la teoría de la planificación. La planificación es un proceso de toma de decisiones donde n trabajos son planificados en m máquinas. Siguiendo la nomenclatura introducida por Graham et al. [29] en el problema se identifican tres campos: α describe el entorno de la máquina, β las características de procesado y sus limitaciones y γ la función objetivo. Cada uno de estos campos puede tomar uno o varios valores.

El lanzamiento de una tarea o trabajo en terminología de planificación en una GPU de NVIDIA normalmente involucra tres tipos de comandos. Un comando de transferencia del *host* al dispositivo (*HtD*), la ejecución de un *kernel* y, finalmente, una transferencia de datos desde el dispositivo al *host* (*DtH*). Pueden existir cero, uno o más de estos comandos en una tarea. Los comandos de transferencia son ejecutados en los motores DMA y el *kernel* en los cores de la GPU. Cuando un *kernel* es cargado en la GPU, los recursos *hardware* (memoria, registros, etc) son asignados a dicho *kernel*. Si un *kernel* no consume todos los recursos disponibles, la ejecución concurrente de *kernels* (CKE) podría llevarse a cabo. Sin embargo, la mayoría de los *kernels* en aplicaciones reales son diseñados para usar todos los recursos. En este capítulo restringimos nuestro análisis a *kernels* que no se ejecutan concurrentemente. Por otro lado, las GPUs modernas poseen dos motores DMA que permiten lanzar comandos de transferencia de forma simultánea en direcciones opuestas. De esta forma podemos considerar tres máquinas en nuestro sistema (dos motores DMA y la GPU). Esto se corresponde con un problema del tipo *Flow Shop*, donde cada trabajo debe realizarse en una máquina siguiendo un orden determinado. Por lo tanto, α es representado por *F3* que describe un problema *Flow Shop* con 3 máquinas. La figura 3.5 muestra un ejemplo con dos trabajos en un problema *Flow Shop* con 3 máquinas.

Una generalización de este problema es el *Flow Shop* flexible donde, en lugar de tener m máquinas en serie, hay c etapas en serie. Cada etapa incluye un conjunto de máquinas idénticas en paralelo que pueden ejecutar el mismo tipo de trabajo. Esta configuración puede usarse para describir un *cluster* de GPUs. Así, podría haber tres etapas (una etapa por cada tipo de comando), con varias máquinas como GPUs en un *cluster* y el problema podría representarse como

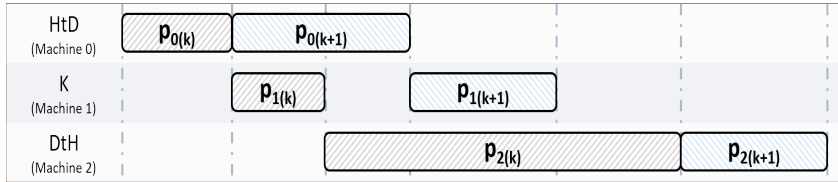


Figura 3.5: Lanzamiento de tareas GPU como un problema *Flow Shop* de 3 máquinas. Los comandos *HtD* son ejecutados por la máquina 0, los comandos *K* por la máquina 1 y los comandos *DtH* por la máquina 2. Los comandos del trabajo k preceden a los comandos del trabajo $(k+1)$ en todas las máquinas (*Flow Shop* con permutación). $p_{i(j)}$ corresponde al tiempo de procesamiento del trabajo j en la máquina i .

FF3.

En cuanto a las características y limitaciones de procesamiento, se pueden tener en cuenta varios valores. Por ejemplo, una característica típica de *Flow Shop* considera que cada máquina opera bajo el supuesto de la política del primero que llega es el primero en servirse (FCFS). Esta política no se mantiene en las GPUs con HyperQ a menos que se utilicen eventos para imponer un orden. Por otro lado, se sabe que, para los sistemas F3, siempre existen planificaciones óptimas que no requieren cambios de secuencia entre máquinas [82] y, en ese caso, se pueden hacer algunas simplificaciones que facilitan la búsqueda de una planificación óptima. Así, en este trabajo, se utilizan eventos para imponer el mismo orden en todas las máquinas. Este tipo de problema *Flow Shop* se denomina *Flow Shop* con permutación y se anota utilizando la palabra *prmu* en el campo β . En el ejemplo de la figura 3.5, se observa que las tareas del trabajo k preceden a las tareas del trabajo $(k+1)$ en todas las máquinas.

En el mundo real, uno o varios procesadores (o hilos) ejecutan parte de su código en la CPU y otra parte en la GPU. Así, las tareas planificadas pueden llegar al sistema en un momento determinado denominado momento de lanzamiento (r_j). Estas tareas no pueden ser lanzadas antes de su momento de lanzamiento y esto puede ser anotado mediante r_j en el campo β . Además, cada procesador podría ejecutar muchas tareas diferentes en la GPU, con algunas restricciones de precedencia entre ellas para asegurar su correcta ejecución. Estas restricciones suelen codificarse mediante un grafo acíclico dirigido y pueden expresarse utilizando la palabra *prec* en β . Por último, las tareas pueden pertenecer a diferentes usuarios, por lo que deben lanzarse en diferentes contextos de la GPU. En la teoría de planificación, esto es equivalente a las familias de trabajos, es decir, los

trabajos de la misma familia pueden procesarse uno tras otro sin ningún retraso, pero cambiar de una familia a otra requiere un tiempo de preparación, s , antes de que se lancen los trabajos. Este tiempo de preparación es el tiempo que se tarda en crear un contexto de la GPU, y en la notación de la teoría de planificación se expresa con $fmls$ en β .

También es posible incluir información sobre las características de la tarea en β , como su tiempo de procesamiento o su prioridad. Por ejemplo, nos referiremos a p_{ij} como el tiempo de procesamiento de la tarea j en la máquina i (como en la figura 3.5, donde $P_{0(k)}$ es el tiempo de procesamiento de la tarea k en la máquina 0 y así sucesivamente). Aunque es posible obtener una estimación del tiempo de procesamiento de cada tarea por adelantado, los tiempos reales de transferencia dependen de si hay otra transferencia en sentido contrario o no [44]. Por lo tanto, estas fuentes de incertidumbre pueden dar lugar a planificaciones subóptimas al resolver la función objetivo. También se puede asignar un factor de prioridad a cada tarea mediante un valor de peso, w_j , que se puede tener en cuenta en la función objetivo. Además, los tiempos de vencimiento d_j , que reflejan el tiempo de finalización de la tarea i , también pueden incluirse en β .

Por último, hay que tener en cuenta la función objetivo que define la planificación óptima. El tiempo de finalización del trabajo j puede denotarse como C_j . La función más común en los problemas *Flow Shop* es minimizar el tiempo de ejecución C_{max} o *makespan*, definido como $\max(C_1, \dots, C_n)$, y equivale a minimizar el tiempo de finalización del último trabajo que sale del sistema. En este tipo de problemas, es casi equivalente a maximizar el uso de las máquinas, que es nuestro principal objetivo. Otra función objetivo útil es el tiempo total de finalización ponderado ($\sum w_j C_j$), también denominado tiempo de flujo ponderado, que tiene en cuenta los factores de prioridad dado por los valores de peso. En la tabla 3.3 se muestra un resumen de las notaciones usadas para los campos α , β y γ .

Campo	Valor	Significado	Descripción
α	Fm	<i>Flow Shop</i> con m máquinas	Cada trabajo se procesa en un orden específico
	FFc	<i>Flow Shop</i> flexible con c etapas	Como Fm pero usando c etapas de máquinas idénticas
	$prmu$	Permutación	Las máquinas trabajan bajo la política FIFO
β	r_j	Momento de salida	Las tareas llegan en un momento en particular
	w_j	Factor de prioridad	Las tareas son asignadas a un factor de ponderación
	d_j	Momento dado	Las tareas son asignadas a un momento dado
	$prec$	Limitaciones de precedencia	Las tareas deben seguir un gráfico acíclico directo
	$fmls$	Familias de trabajo	Las tareas provienen de distintos usuarios
γ	C_{max}	Makespan	Minimizar el tiempo de ejecución de la última tarea
	$\sum w_j C_j$	Tiempo total de finalización ponderado	El factor de prioridad se incluye

Tabla 3.3: Resumen de las notaciones para la teoría de la planificación.

3.5. Heurísticas de Planificación

El problema presentado en este capítulo es común en muchas situaciones en las que uno o varios hilos lanzan varias tareas que pueden ser calculadas utilizando el mismo contexto de la GPU. Por ejemplo, en una aplicación de videovigilancia, se podrían usar varios hilos que analicen diferentes canales de vídeo y parte del cálculo podría asignarse a una GPU para acelerar todo el proceso. De forma similar al servicio MPS (*MultiProcess-Service*) [60] de CUDA, se podría utilizar un hilo *proxy* para planificar todas las tareas y lanzarlas utilizando el mismo contexto de la GPU para mejorar la concurrencia entre las tareas. En el caso de que queramos minimizar el tiempo de ejecución, este problema es equivalente a $F3||C_{max}$ y es de tipo NP-hard. De hecho, para un problema en el que se quieran procesar N tareas, hay $N!$ permutaciones diferentes.

Se han presentado muchas heurísticas para resolver eficientemente este problema [84]. Estas heurísticas se pueden clasificar como constructivas o de mejora, dependiendo de si replantean el problema desde cero o mejoran una solución anterior. En este trabajo, estamos interesados en la planificación en tiempo real; por lo tanto, consideraremos sólo algoritmos ligeros excepto una heurística que consume mucho tiempo y que se ejecutará *off-line*. Con el fin de poder realizar una comparación adecuada hemos incluido algunas heurísticas clásicas que no tienen en cuenta el efecto producido al solapar dos transferencia de datos en sentido opuesto, otra heurística desarrollada específicamente para la ejecución eficiente de tareas en la GPU y una nueva que combina una heurística de planificación clásica con un modelo de ejecución de tareas en la GPU. En un principio otras dos heurísticas clásicas fueron consideradas, *Modified Johnson's Rule* (MJR) [38] y *Mixed Integer Programming* (MIP) [96], que finalmente se descartaron porque sus resultados eran similares a los de las otras heurísticas clásicas y, en el caso de MIP, su coste computacional la hacía inútil en una aplicación en tiempo real.

3.5.1. Slope index

En la literatura se han propuesto algunas heurísticas que tienen en cuenta un número arbitrario de máquinas como, por ejemplo, la heurística *Slope Index* [77], para encontrar planificaciones cuasi-óptimas para el problema *Flow Shop*. Esta heurística da prioridad a las tareas que tienen la mayor tendencia a progresar de tiempos cortos a tiempos largos en la secuencia de procesos. En nuestro problema de GPU, esto significa que las tareas con tiempos HtD cortos y tiempos DtH largos tienen prioridad sobre las tareas con tiempos HtD largos y tiempos DtH cortos. Esta prioridad se establece calculando un índice de pendiente, *Slope Index*,

A_j para el trabajo j como $A_j = -\sum_{i=1}^m (m - (2i - 1))p_{ij}$, donde m es el número de máquinas (3 en nuestro problema de GPU), y los trabajos se secuencian en orden decreciente de este índice. Nos referiremos a esta heurística como SI (*Slope Index*).

3.5.2. NEH heuristic

Nawaz et al [56] desarrollaron un algoritmo para resolver el problema *Flow Shop*, que se considera una de las mejores heurísticas de planificación. Este algoritmo se conoce como NEH y es una heurística constructiva que itera para obtener una planificación.

1. Para cada tarea se calcula su tiempo total de ejecución y se ordenan en orden decreciente.
2. Se eligen las dos primeras tareas y se encuentra el mejor orden de lanzamiento calculando el tiempo de ejecución para ambas secuencias.
3. Para el resto de tareas, $i = 3, \dots, n$ se encuentra la mejor planificación colocándolas en todas las posiciones posibles i en la secuencia de tareas que ya están planificadas y calculando el *makespan*.

La mayoría de las heurísticas intentan planificar primero las mejores tareas, es decir, las más cortas, pero esta heurística comienza probando las peores tareas (más largas) y acomoda las tareas restantes para minimizar los tiempos de ejecución. El mayor inconveniente es que el *makespan* debe calcularse $[n(n + 1)/2] - 1$ veces, de los cuales n son secuencias completas y el resto son planificaciones parciales, pero a cambio puede obtener mejores resultados que otras heurísticas [91].

3.5.3. Single queue

El enfoque anterior asume que todos los tiempos de ejecución son fijos pero, de hecho, dependen de los comandos que se están procesando en cada momento. En concreto, si un comando *HtD* se ejecuta al mismo tiempo que un comando *DtH*, sus tiempos de procesamiento cambiarán [44]. Esto se debe a que, aunque el bus PCIe es bidireccional y las GPU modernas tienen dos motores DMA, el sistema de memoria se comparte entre ambas operaciones de copia y se reduce el ancho de banda de los comandos *HtD* y *DtH*. Un modelo de transferencia

que no considere la superposición puede incurrir en errores superiores al 10%. Por lo tanto, es probable que estas heurísticas anteriores no logren obtener una planificación óptima.

En el trabajo de Lázaro-Muñoz et al [44] se presentó un modelo de ejecución de tareas en GPU y una heurística basada en un modelo de transferencias superpuestas. Ese modelo utiliza una estimación de los comandos HtD , K y DtH de cada tarea para simular la ejecución de una permutación de las tareas. Los tiempos HtD y DtH se actualizan *on-line* para reflejar el efecto de las transferencias superpuestas, logrando un error de simulación promedio por debajo del 1,5%. Sin embargo, para obtener una estimación correcta, Hyper-Q debe ser restringido. Cualquier cambio en el orden original realizado por Hyper-Q podría introducir un error significativo en la simulación; por lo tanto, Hyper-Q se deshabilita forzando una sola cola administrada por *hardware*.

La heurística presentada en ese trabajo utiliza el modelo de ejecución para realizar una simulación *on-line* de la ejecución para elegir las tareas que mejor se superponen en cada instante. El razonamiento detrás de la heurística implementada en ese trabajo es similar a *Slope Index*, pero utiliza el modelo de ejecución de tareas para seleccionar, de manera iterativa, la tarea que mejor se adapta a los comandos de tareas actuales. Las tareas con comandos HtD cortos se seleccionan primero, mientras que las tareas siguientes se eligen buscando la que mejor se ajuste entre los comandos K restantes de las tareas seleccionadas anteriormente y el comando HtD de la nueva tarea, y entre los comandos DtH restantes de las tareas seleccionadas previamente y el comando K de la nueva tarea. Nos referiremos a esta heurística como SQ (Single Queue).

3.5.4. NEH heuristic para ejecución en GPU

En la heurística NEH original, el *makespan* se calcula utilizando un tiempo de ejecución fijo para cada comando, pero en una GPU el tiempo de ejecución de dos comandos de transferencia superpuestos puede variar significativamente con respecto a su ejecución por separado, como se mostró en la sección 3.3.1. Por tanto, el *makespan* predicho puede tener un gran error que puede llevar a una planificación inadecuada. En esta tesis, proponemos combinar la heurística NEH con un modelo de ejecución de tareas en la GPU que pueda predecir con mayor exactitud el *makespan*. Nos referiremos a esta heurística como NEH-GPU.

Para mostrar las ventajas de utilizar el modelo de ejecución propuesto en esta tesis recurrimos de nuevo a la figura 1.2. Esta figura compara los resultados obtenidos por NEH original (*Permutation I*) con NEH-GPU (*Permutation II*) en

la planificación de cuatro tareas independientes, estas son *Matrix Multiplication* (MM), *Black-Scholes* (BS), *Matrix Transposition* (TM) y *Vector Addition* (VA). Para cada tarea se da, en la tabla 3.4, la duración de sus comandos *HtD*, *K* y *DtH*, y su *makespan* total si se lanzan solas. Además, en la tabla 3.5, se muestran los *makespans* calculados por cada heurística. Las dos tareas más largas, 3 y 0, se prueban en la primera iteración. Cada algoritmo calcula el *makespan* utilizando los datos de la tabla 3.4 y, aunque predicen un *makespan* diferente para la permutación 3 - 0, seleccionan la misma permutación, 0 - 3, como la que tiene el mejor *makespan* parcial. En la segunda iteración, prueban todas las permutaciones posibles insertando la tarea 2 y predicen diferentes *makespans* que les llevan a seleccionar diferentes permutaciones, es decir, 0 - 3 - 2 para NEH y 2 - 0 - 3 para NEH-GPU. Finalmente, en la última iteración, cada heurística inserta la tarea restante obteniendo diferentes planificaciones. La heurística NEH predice un *makespan* de 27,32 al planificar 0 - 3 - 2 - 1, pero la ejecución real tarda 29,79. Por otro lado, la heurística propuesta en esta tesis (NEH-GPU) predice un *makespan* de 26,58 para la planificación 1 - 0 - 3 - 2 y la ejecución real tarda 26,97. Es decir, el modelo de la GPU predice con mucha precisión el tiempo de ejecución real, lo que lleva a una planificación óptima de estas tareas.

	Tarea	HtD	K	DtH	Total
0	MM	2,52	10,61	1,26	14,39
1	BS	0,73	8,50	0,49	9,72
2	TM	5,03	0,31	4,98	10,32
3	VA	10,04	0,37	4,98	15,39

Tabla 3.4: Tiempo de ejecución de los comandos de las tareas MM, BS, TM y VA

Paso	Orden	NEH	NEH-GPU
1º	3-0	24,43	25,32
	0-3	19,37	19,37
	2-0-3	24,40	25,57
2º	0-2-3	24,35	26,26
	0-3-2	24,35	26,26
	1-0-3-2	31,06	26,58
3º	0-1-3-2	32,08	31,28
	0-3-1-2	27,47	33,04
	0-3-2-1	27,32	29,30

Tabla 3.5: *Makespan* usando NEH y NEH-GPU. Orden testeado y *makespan* predicho usando cada heurística.

3.6. Experimentos

Los experimentos de esta tesis se han realizado utilizando un conjunto de *kernels* reales obtenidos de las *benchmarks* suite de CUDA [66] y Rodinia [18]. Se han seleccionado tareas con diferentes tiempos de transferencia y procesamiento del kernel, resumidos en la tabla 3.6. Hay tareas con un comando *HtD* corto y un comando *K* largo, tareas con comandos *HtD* y *DtH* largos, y comandos *K* cortos, etc. Aunque algunas de estas tareas tienen varios comandos del mismo tipo (por ejemplo, *Matrix Multiplication* debe transferir dos matrices del *host* al dispositivo, o *PathFinder* lanza varios comandos del *kernel* en orden), el planificador considera que hay un único comando que engloba todos los comandos del mismo tipo.

Kernel	Fuente	Descripción	Categoría
MM	CUDA SDK	Matrix Multiplication	DK
BS	CUDA SDK	Black Scholes	DK
PF	Rodinia	Path Finder	DK
PAF	Rodinia	Particle Filter	DK
CONV	CUDA SDK	Separable Convolution	DK/DT
VA	CUDA SDK	Vector Addition	DT
TM	CUDA SDK	Matrix Transposition	DT
FWT	CUDA SDK	Fast Walsh Transform	DT

Tabla 3.6: Tareas usadas en los experimentos CTE. Las tareas han sido clasificadas en Kernel Dominante (DK) o Transferencia Dominante (DT) dependiendo de la duración de los comandos *HtD* y *DtH*. La tarea CONV puede ser DK (CONV1) o DT (CONV2) dependiendo de sus parámetros de entrada.

Para representar una carga computacional más variada, se han seleccionado diferentes parámetros de entrada para aumentar el número de tareas a 21. Además, se han considerado tres arquitecturas de GPU, una K20c (Kepler), una GTX 980 (Maxwell) y una Titan X (Pascal). Por último, para evitar el efecto de los valores atípicos, cada experimento se ejecuta quince veces para registrar un *makespan* medio.

3.6.1. Análisis estadístico

En este apartado se analiza estadísticamente nuestro modelo de ejecución de tareas concurrentes para evaluar la validez de la nueva heurística y compararla con las anteriores. Con este objetivo, hemos obtenido todas las combinaciones po-

sibles de cuatro tareas, con repetición, del conjunto de 21 tareas. Esto representa un total de $\binom{21}{4} = 10.626$ combinaciones de cuatro tareas cada una. Para cada combinación, hay $4! = 24$ planificaciones diferentes; por lo tanto, hay 255.024 experimentos diferentes que se han ejecutado 15 veces para registrar el *makespan* medio para cada una. Hemos seleccionado la mediana y el mínimo del *makespan* para cada combinación, así como el tiempo de ejecución obtenido por cada heurística. Por último, hemos calculado el *speedup* de cada heurística con respecto a la mediana de la combinación y, a efectos de comparación, el *speedup* del mínimo con respecto a la mediana (mejor *speedup*). Estos valores se representan en la figura 3.6a utilizando gráficas de caja para visualizar sus propiedades estadísticas [93]. En cada caja, la marca central corresponde a la mediana, los bordes de la caja son los percentiles 25 y 75, los bigotes se dibujan con líneas discontinuas y se extienden hasta los puntos de datos más extremos no considerados como valores atípicos (alrededor de $\pm 2,7\sigma$ y una cobertura del 99,3%), los valores atípicos se representan individualmente con signos “+”, y hay muescas en las marcas de la mediana para los intervalos de comparación. Dos medianas son significativamente diferentes al nivel de significación del 5% si sus intervalos no se solapan.

La caja más a la izquierda de la Figura 3.6a corresponde a la heurística de *Slope Index* (SI). Esta heurística es muy sencilla, pero los resultados son pobres. Así, su valor medio está muy cerca de 1 (es decir, casi no se obtiene *speedup*) y en muchas combinaciones el *speedup* es inferior a 1. La siguiente caja muestra los resultados del algoritmo NEH. Los resultados son mucho mejores, con una mediana de *speedup* cercana a 1,05, y la mayoría de los *speedups* están por encima de 1, aunque el bigote inferior se extiende por debajo de 0,9. La caja del centro corresponde a la heurística de *Single Queue* (SQ). Los resultados son mejores que los de SI, pero no tan buenos como los de NEH. La mayoría de los valores de *speedup* están por encima de 1 y el bigote inferior se extiende por debajo de 0,9 como en el algoritmo NEH. Finalmente, los dos últimos recuadros corresponden al nuevo algoritmo presentado en este capítulo (NEH-GPU) y al mejor *speedup* alcanzable (*Best*). NEH-GPU obtiene resultados muy cercanos a los mejores, con una mediana ligeramente inferior y el bigote inferior se extiende por encima de 0,9. En comparación con las otras heurísticas, su valor mediano de aceleración es mayor y no tiene valores atípicos por debajo del bigote inferior. Todas las cajas son muy estrechas, pero ninguna de ellas se solapa; así, cada heurística es significativamente diferente al nivel de significación del 5%. Lo hemos confirmado utilizando un t-test entre los valores de *speedup* de NEH y NEH-GPU para descartar que pertenezcan a la misma distribución.

Tanto los resultados de NEH-GPU como los de *Best* muestran que hay muchas combinaciones en las que se pueden obtener *speedups* de hasta 1,40, y sería

interesante determinar qué tipo de experimentos pueden obtener más beneficios de una buena planificación. Un enfoque sensato podría ser estudiar las oportunidades de solapamiento entre varias tareas; por ejemplo, las tareas con comandos *K* largos se pueden solapar fácilmente con tareas con comandos *HtD* largos. Por tanto, hemos clasificado las tareas como transferencia dominante (DT) o kernel dominante (DK) según predomine el tiempo de transferencia sobre el tiempo del kernel, o al revés, y hemos analizado los resultados obtenidos por diferentes combinaciones de DT y tareas DK. La Figura 3.6b muestra los diagramas de caja de los valores de *speedup* obtenidos por el algoritmo NEH-GPU separando los resultados por el número de tareas DK. La caja más a la izquierda corresponde a experimentos donde no hay tareas DK (las cuatro tareas son DT), la siguiente caja tiene en cuenta experimentos con solo una tarea DK y tres tareas DT, y así sucesivamente. Se puede ver que los experimentos en los que todas las tareas son del mismo tipo obtienen pocos beneficios de una buena planificación, pero los experimentos con una combinación equilibrada de tareas DK y DT tienen más oportunidades de solaparse y los valores de *speedup* pueden ser mucho más altos.

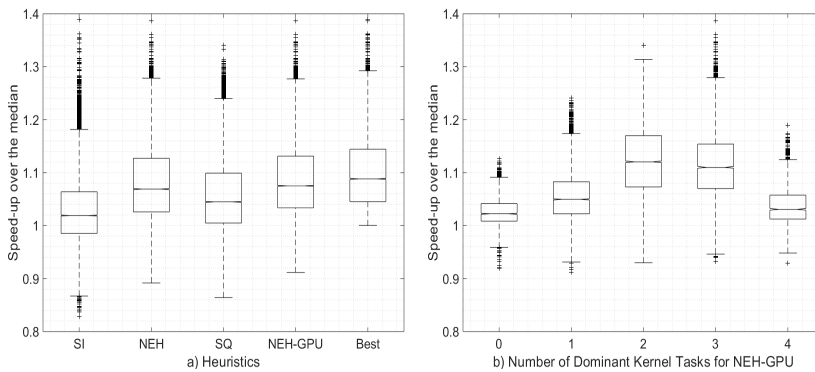


Figura 3.6: Diagrama de cajas de *speedup* sobre *makespan* medio de 10.626 combinaciones de 4 tareas en K20c, GTX 980 y Titan X.

3.6.2. Aplicabilidad y Escalabilidad

En esta sección, las cuatro heurísticas se evalúan en un escenario multi-hilo exigente (por ejemplo, un sistema heterogéneo) donde varios hilos ejecutan aplicaciones (trabajadores) mandando tareas a un dispositivo. Todos los trabajadores envían información de la tarea a un *buffer* que un hilo *proxy* en el *host* sondea constantemente. Esta información incluye la identificación del trabajador y las

llamadas de lanzamiento del *kernel*, *HtD* y *DtH* de la API de CUDA. El hilo *proxy* se encarga de reordenar el conjunto de tareas que se encuentran en el *buffer* utilizando una de las cuatro heurísticas y envía los comandos correspondientes al dispositivo.

La escalabilidad de cada heurística se prueba aumentando el número de tareas de 4 a 8 y 16. Este número de tareas representa un gran número de combinaciones diferentes, $\binom{21}{8} \approx 310^6$ para ocho tareas y $\binom{21}{16} \approx 10^9$ para dieciséis tareas. Además, $8! = 40.320$ planificaciones posibles con ocho tareas y $16! \approx 10^{13}$ con dieciséis tareas. Por tanto, se ha seleccionado un subconjunto aleatorio de diferentes *benchmarks* para cada combinación de número de tareas (4, 8 y 16) y las arquitecturas de GPU (K20c, GTX 980 y Titan X), y se han obtenido los resultados para cada heurística. A efectos comparativos, se ha registrado el mejor *makespan* (BM) obtenido para cada combinación en cada arquitectura para establecer un valor mínimo de referencia. Luego, para cada experimento, el *makespan* obtenido por cada heurística (HM) se compara con este valor de referencia para evaluar el valor de proximidad. Esta proximidad se calcula dividiendo ambos valores ($BM/HM \cdot 100$) para obtener un porcentaje con un valor máximo de 100% (cuanto mayor, mejor). Los resultados incluyen el *overhead* del cálculo de la planificación incurrido por cada heurística. Esta sobrecarga va desde $1\mu s$ para SI hasta menos de $1ms$ para el resto de heurísticas cuando se planifican dieciséis tareas. Por lo tanto, teniendo en cuenta la duración del *makespan*, el *overhead* de la planificación es insignificante.

La figura 3.7 muestran la media de los resultados obtenidos con estos experimentos para cada heurística. Las conclusiones dependen del número de tareas, la planificación de tareas de kernel dominante (DK) y las diferentes arquitecturas de GPU, respectivamente. Un valor de proximidad cercano al 100% significa que la heurística obtiene el mejor *makespan*, o un valor muy cercano a él, la mayoría de las veces. Se puede observar que la nueva heurística NEH-GPU obtiene consistentemente mejores resultados de proximidad que las otras heurísticas en todas las situaciones y arquitecturas GPU.

Existe una limitación, impuesta por CUDA, sobre la cantidad de tareas que se pueden ejecutar simultáneamente en una GPU. Esta limitación se debe a la sincronización implícita provocada por cualquier comando de liberación de la memoria del dispositivo. En concreto, cuando se llama a *cudaFree* desde cualquier tarea, todos los comandos CUDA anteriores en ese contexto deben finalizar antes de que se inicie cualquier comando nuevo. Por lo tanto, todas las tareas que se ejecutan simultáneamente deben caber en la memoria del dispositivo. Las tarjetas GPU utilizadas en este trabajo tienen un tamaño de memoria de entre 4 GB y 12 GB, mientras que la mayoría de los *kernels* de la tabla 3.6 necesitan entre

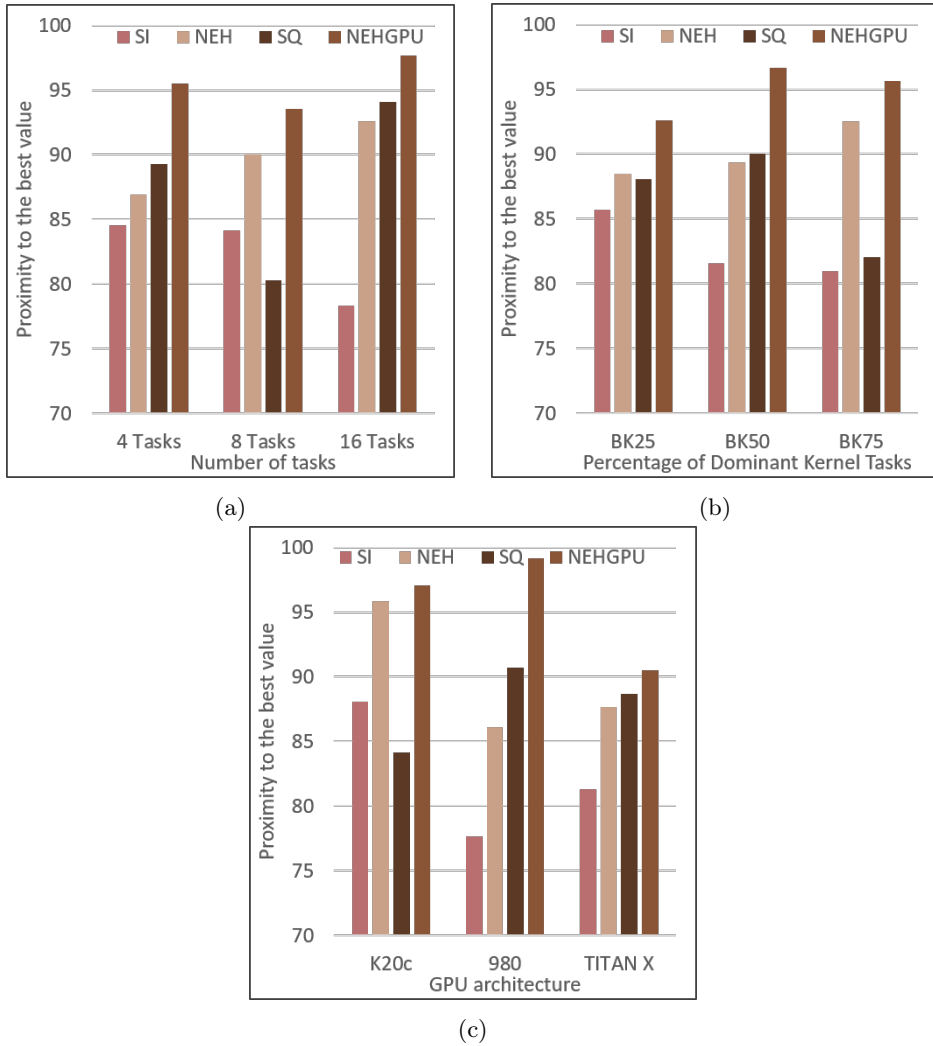


Figura 3.7: Proximidad al mejor resultado (a) con un número diferente de tareas, (b) con distintas proporciones de tareas *kernel* dominante y (c) con distintas arquitecturas de GPU

1 MB y 128 MB, según los parámetros de entrada. Por lo tanto, una cantidad máxima razonable de tareas que se ejecutan simultáneamente podría ser 64 para la mayoría de las arquitecturas NVIDIA actuales. Sin embargo, este número de

tareas es demasiado alto para ser planificado por nuestra heurística en un tiempo razonable; por lo tanto, a continuación, presentaremos un método rápido y eficiente para resolver este problema.

A partir de los experimentos realizados, se puede ver, por un lado, que los mejores resultados se obtienen cuando se planifican una tarea DK y una tarea DT y, por otro lado, se pueden planificar hasta 16 tareas con un *overhead* bajo. Por tanto, en lugar de calcular la planificación de las 64 tareas juntas, agruparemos estas tareas en cuatro lotes buscando el mejor equilibrio entre las tareas DK y DT, y calcularemos la planificación de cada lote por separado.

En el siguiente experimento, consideramos que hay cientos o miles de tareas que esperan ser ejecutadas en un *cluster* de GPUs. Para maximizar la utilización de la GPU, estas tareas se agrupan en conjuntos de 64 tareas que se lanzan en cada GPU. Simularemos esta situación seleccionando al azar 64 tareas de la tabla 3.6 y lanzándolas sin ningún orden en particular. Luego, usamos el método anterior con las heurísticas NEH y NEH-GPU para obtener otra planificación, y se compararán sus tiempos de ejecución. Hemos repetido esto 10^5 veces en cada una de nuestras tarjetas GPU para obtener diagramas de caja de la aceleración tanto de NEH como de nuestra heurística con respecto al planificador aleatorio. Estos diagramas de caja se muestran en la figura 3.8, donde los valores medios de NEH-GPU son 1,2630 para K20c, 1,2050 para GTX 980 y 1,2468 para Titan X. Estos experimentos utilizan subconjuntos aleatorios de tareas; por lo tanto, las aceleraciones son muy variables. No obstante, los valores en el segundo y tercer cuartil son siempre superiores a 1 en todas las tarjetas GPU y mejores que las aceleraciones obtenidas por NEH.

3.6.3. Comparación con MPS

CUDA proporciona MPS, una herramienta que permite que las aplicaciones multi-hilo desarrolladas para su ejecución en la GPU de NVIDIA utilicen las capacidades de Hyper-Q. Con MPS, las aplicaciones CUDA comparte un contexto de GPU creado por el proceso servidor MPS. Por lo tanto, MPS puede coejecutar la transferencia de datos de GPU y los comandos de ejecución del kernel lanzados por diferentes aplicaciones. Como nuestra propuesta también aprovecha esta concurrencia de comandos para reducir el *makespan*, llevamos a cabo un experimento comparativo en esta sección. En concreto, comparamos el rendimiento que consigue MPS cuando varios procesos lanzan tareas de GPU con el obtenido por nuestro planificador.

A la izquierda de la figura 3.9 se muestra una captura de la salida de la he-

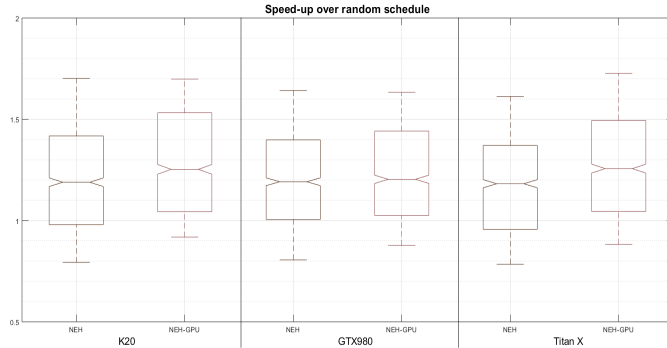


Figura 3.8: Diagrama de cajas de speedup para las heurísticas NEH y NEH-GPU sobre la planificación aleatoria cuando se cargan 64 tareas en K20c, GTX 980 y Titan X.

ramienta *NVIDIA Visual Profiler*, durante la inicialización de cuatro tareas por sus hilos correspondientes (dos MM, con diferentes tamaños en sus matrices, un FWT y un VA). Los cuadros de color naranja oscuro corresponden a las llamadas a la API de CUDA, mientras que los otros cuadros de colores corresponden a los comandos *HtD*, *kernel* y *DtH* que se ejecutan en el mismo contexto de CUDA (administrado por MPS) en una GPU Tesla K20c. Para hacer una comparación justa con nuestro *proxy*, todas las aplicaciones CUDA se sincronizan, utilizando semáforos POSIX, antes de iniciar su primera transferencia del *host* al dispositivo. Esta sincronización intenta evitar penalizaciones de planificación introducidas por el SO durante la ejecución de las aplicaciones. Finalmente, el *makespan* se calcula midiendo el tiempo desde el comienzo del primer comando *HtD* hasta el final del último comando *DtH*. Como resultado secundario del uso de MPS, ahora los comandos *kernel* pueden superponerse parcialmente, como se muestra en la figura 3.9, con una tarea MM y otra FWT.

Hemos comparado MPS con los resultados obtenidos por NEH-GPU utilizando un hilo de servidor *proxy* como en la sección 3.6.2. Al igual que en el primer experimento, se han ejecutado 10.626 combinaciones distintas de cuatro tareas quince veces en cada tarjeta GPU para obtener el *makespan*. A veces, el sistema operativo introduce una penalización severa que degrada seriamente el rendimiento de MPS; por lo tanto, hemos eliminado todos estos valores anormales utilizando la desviación mediana absoluta [48]. En la parte derecha de la figura 3.9 muestra un diagrama de cajas de la aceleración de nuestra heurística con respecto a MPS. Se puede ver que, aunque hay algunos resultados en los que

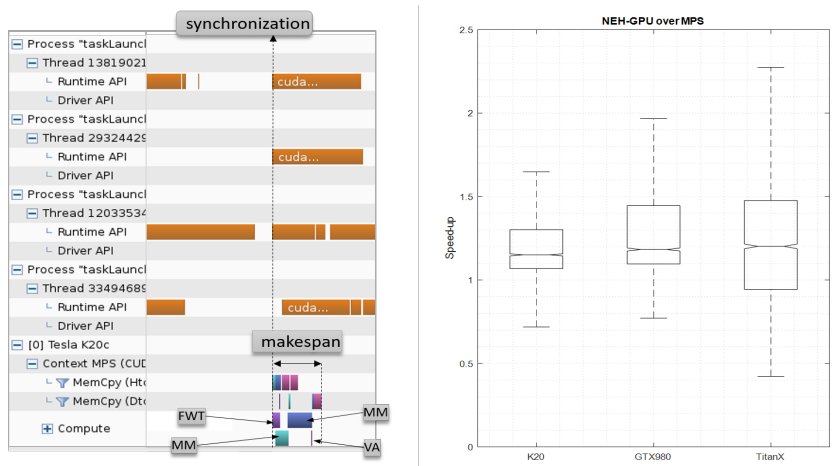


Figura 3.9: Comparativa entre MPS y NEH GPU. La figura muestra la salida del *profiler* para el proceso de lanzamiento de cuatro tareas diferentes (dos MM, una FWT y una VA) ejecutadas en el mismo contexto de CUDA y gestionadas por el proceso servidor MPS. La figura de la derecha muestra un diagrama de cajas del *speedup* de NEH-GPU sobre MPS.

NEH-GPU funciona peor que MPS, la mayoría de los valores están por encima de 1. Los valores por debajo de 1 generalmente corresponden a combinaciones donde dos o más comandos *kernel* pueden solaparse, mientras que los valores por encima de 1 generalmente se deben a una mejor planificación de las cuatro tareas por parte de NEH-GPU. La mediana de aceleración de NEH-GPU sobre MPS es 1,1509 para K20c, 1,1830 para GTX 980 y 1,2018 para Titan X. Estos resultados confirman los beneficios de usar una heurística de planificación eficiente cuando se ejecutan tareas independientes concurrentemente en la GPU.

3.7. Resumen

En este capítulo se ha presentado un modelo que simula la ejecución de varias tareas en una GPU. Este modelo se puede utilizar para encontrar el orden de ejecución que permite obtener un tiempo de ejecución menor. Existen varios trabajos previos que estudian las políticas de planificación de tareas en las GPU pero sin tener en cuenta la teoría general de planificación. En esta tesis se aplican los conceptos de dicha teoría a este problema, demostrando que la ejecución

concurrente de tareas en una GPU utilizando *streams* de CUDA se puede modelar como un problema de tipo *Flow Shop*. La ventaja más importante de este enfoque es que se puede definir una función objetivo y seleccionar una solución adecuada dentro de la literatura existente. Como ejemplo práctico, se ha estudiado el problema $F3|prmu|C_{max}$ que surge cuando varios hilos lanzan *kernels* independientes que pueden ser computados usando el mismo contexto GPU.

Se han presentado varias soluciones existentes en el estado del arte para la planificación de tareas en la GPU (heurísticas SI, NEH y SQ). Además, se ha desarrollado una nueva heurística denominada NEH-GPU, que combina una heurística existente (NEH) con un modelo de ejecución de tareas en GPU. Esta heurística también se puede incluir como soporte en tiempo de ejecución porque no modifica los *kernels* originales y tiene un *overhead* reducido. El modelo de ejecución de tareas de GPU incluye un modelo de transferencia de datos preciso que predice el tiempo de ejecución de cada comando de transferencia de datos. Este modelo puede considerar diferentes arquitecturas de GPU (Kepler, Maxwell y Pascal), tipos de memoria (paginable y *pinned*) y transferencias de datos simultáneas.

Se han realizado varios experimentos para demostrar la idoneidad y solidez de este modelo. Se han evaluado tres arquitecturas de GPU diferentes: Kepler, Maxwell y Pascal, utilizando varios *kernels* reales de CUDA y Rodinia SDK. Se ha realizado un análisis estadístico para evaluar la importancia de la heurística NEH-GPU y mostrar su ventaja sobre otras heurísticas. Además, se ha variado el número de tareas para evaluar la escalabilidad de las heurísticas. En todos ellos, NEH-GPU ha obtenido los resultados más cercanos al mejor *makespan* obtenido por cualquier heurística. Finalmente, se ha realizado una comparación con MPS que muestra que nuestro planificador obtiene aceleraciones que van desde 1,15 a 1,20 sobre la solución proporcionada por Nvidia.

Creemos que este enfoque podría incluirse en el servidor MPS o como soporte en tiempo de ejecución de cualquier GPU para mejorar la concurrencia entre los comandos de *kernel* y de transferencia (*HtD* y *DtH*).



UNIVERSIDAD
DE MÁLAGA

4 Ejecución concurrente de kernels mediante métodos software

En este capítulo se aborda el problema de la ejecución concurrente de *kernels* (CKE) aplicando métodos *software*. Estos métodos buscan planificar un conjunto de *kernels* para su coejecución mediante una distribución adecuada de los bloques de los distintos *kernels* sobre los *Streaming Multiprocessors* (SMs) que permita mejorar el uso de estos recursos *hardware* y reducir el tiempo de ejecución del conjunto de *kernels* coejecutados. Para evitar el impacto de las transferencias, en este análisis asumimos que los *kernels* están listos para ser lanzados simultáneamente, es decir, todos los comandos previos al *kernel* han terminado su ejecución y existen *kernels* pendientes de ser ejecutados.

Para resolver este problema hemos desarrollado un planificador *software* que incluye un método de *profiling* productivo *on-line* que ayuda a encontrar la mejor partición de recursos del SM cuando se inicia la ejecución concurrente. Éste utiliza un mecanismo flexible que es capaz de gestionar el desalojo y el lanzamiento parcial del *kernel*. Además, con estos métodos es posible la aplicación de políticas de planificación como por ejemplo asegurar cierta calidad de servicio (*Quality of Service, QoS*).

En la sección 4.1 se analiza el estado del arte referente a la ejecución concurrente de *kernels* en la GPU aplicando mecanismos *software*. Los dos mecanismos más habituales, *spatial multitask* y *simultaneous multikernel*, son presentados en la sección 4.2. En la sección 4.3 se describe la motivación por la cual se plantea nuestro mecanismo *software*, mientras que la sección 4.4 presenta nuestro modelo

software, denominado *FlexSched*, y se describe su arquitectura. En la sección 4.5 se definen cada uno de los componentes que conforman *FlexSched*. En la sección 4.6 se muestran los experimentos realizados para validar nuestro modelo. Finalmente, en la sección 4.7 se hace un resumen de las conclusiones obtenidas para nuestro modelo *software*.

4.1. Estado del Arte

Las GPUs modernas, desde la arquitectura *Fermi* de NVIDIA, disponen de un soporte *hardware* para la ejecución concurrente de *kernels* (CKE). Este soporte *hardware* hace uso de colas *software*, denominadas *streams* en la terminología de CUDA, para lanzar *kernels* independientes que podrían ejecutarse de forma concurrente en la GPU. Sin embargo, no existe ningún mecanismo a nivel *software* para seleccionar cómo se distribuyen los *Cooperative Thread Array* (CTAs) o bloques de hilos de un *kernel* en la GPU.

En las arquitecturas actuales el planificador *hardware* de CTAs se encarga de lanzar los *kernels*. Este planificador intenta equilibrar la carga dentro de los SMS al lanzar un *kernel* asignando CTAs consecutivos a SMs consecutivos siguiendo un esquema *Round-Robin* (RR) [75]. La planificación no es muy flexible porque sigue una política *leftover* [46]. Así, si un *kernel* lanza más CTAs de los que pueden asignarse simultáneamente en la GPU, no se puede realizar una ejecución concurrente con el siguiente *kernel* planificado hasta que los últimos CTAs del *kernel* en ejecución liberen los recursos de la GPU. Por otro lado, si dos *kernels* tienen un número reducido de CTAs, el planificador *hardware* sí puede ejecutarlos de forma simultánea, pero el planificador no puede controlar su asignación, lo que dificulta el desarrollo de políticas de grano fino que exploten la heterogeneidad disponible de los *kernels*. Para sacar una ventaja real, la ejecución concurrente mediante *software* suele requerir la modificación del código fuente del *kernel*. Se han estudiado alternativas a la política actual del planificador *hardware* empleando simuladores [98, 106]. Estos trabajos proporcionan un control más preciso para la asignación de CTAs, pero no pueden utilizarse en un sistema real.

En este campo se han publicado multitud de trabajos que pretenden mejorar la forma en que se planifican los *kernels* en la GPU. Estos buscan distintos objetivos, como reducir el *makespan* de un conjunto de *kernels* [44, 99], mejorar el tiempo de ejecución durante la coejecución [109] o realizar una planificación de *kernels* basada en prioridades [40]. En trabajos previos [19, 45, 50, 76, 100, 104] se aprecia que algunos *kernels* no escalan de forma adecuada cuando se le asignan un mayor número de recursos *hardware*, debido a la saturación de dichos recursos *hardware*.

En estos trabajos se busca mejorar el uso de recursos mediante la coejecución de *kernels* complementarios, es decir, que hagan uso de distintos recursos *hardware* de la GPU.

Algunos trabajos se han centrado en mejorar el uso del ancho de banda entre los *cores* de la GPU y la memoria. Estos han demostrado que el ancho de banda puede ser un cuello de botella para aplicaciones GPU que hacen un uso intensivo de la memoria [36], [37], [81] y [95]. Por ello, plantean arquitecturas de procesamiento en memoria (PIM) en la GPU para evitar el cuello de botella en memoria y mejorar el rendimiento de las aplicaciones que hacen un mayor uso de este recurso [80], [31]. N. Chatterje et al. [17] han demostrado que aplicar políticas de planificación para aumentar el ancho de banda a memoria puede provocar una gran latencia en el servicio de los *warps* de un bloque. Esto bloquea la entrada de nuevos bloques y reduce la capacidad de computo de las GPUs. Las arquitecturas PIM en GPU propuestas poseen distintos tipos de *cores*, donde ciertos *cores* se encuentran alojados cerca de la memoria, lo que incrementa su ancho de banda con la misma; mientras que el resto de *cores* poseen una conexión habitual con la memoria, por lo que el ancho de banda no se ve incrementado. Este tipo de arquitectura mejora el rendimiento de los *kernels* que hacen un uso intensivo de la memoria y requieren que los datos que van a utilizar sean servidos lo antes posible. Aplicar políticas *software* para CKE en este tipo de arquitectura puede aportar ciertas mejoras en este campo.

Un gran número de investigaciones han presentado planificadores en tiempo de ejecución que aprovechan las ventajas de CKE con el fin de mejorar la utilización del *hardware* de la GPU y desarrollar diferentes políticas de planificación concurrente. Pai et al. [76] han transformado los *kernels* en elásticos. De este modo, reducen el número de CTAs que requiere el *kernel* original y mantienen los CTAs por debajo del número máximo de CTAs residentes por SM. En consecuencia, los recursos libres de la GPU quedan disponibles para los CTAs de otros *kernels* elásticos, permitiendo que varios *kernels* puedan coejecutarse. Un problema importante de esta técnica es que no puede aplicarse a *kernels* que utilizan memoria compartida. Esto se debe a que los accesos a memoria compartida están asociadas a bloques de hilos físicos y no a bloques de hilos lógicos por lo que la modificación en el *kernel* podría dar lugar a comportamientos erróneos. Además, han definido varias políticas de asignación de CTAs, pero todas ellas son estáticas y se basan en modelos que sólo tienen en cuenta el uso de los recursos de la GPU extraídos durante la compilación del *kernel*, en lugar de la información obtenida durante la coejecución de los mismos. En consecuencia, el rendimiento real alcanzado podría ser inferior al original. En [85, 109] han propuesto soluciones para mejorar la coejecución de varios *kernels* mediante el troceado (*slicing*)

de los *kernels*. Cada *slice* computa un pequeño número de bloques y, de este modo, puede coejecutarse con un *slice* de otro *kernel*. El principal problema de este mecanismo es que los *kernels* grandes pueden provocar una gran sobrecarga de lanzamientos, ya que los *kernels* deben dividirse en muchos *slices*.

También han explorado la planificación concurrente de *kernels* por prioridad. Así, Lee et al. [45] han propuesto un mecanismo de planificación basado en prioridades. Dado que este trabajo no implementa técnicas de *preemption* para desalojar un *kernel* en ejecución, el planificador sólo puede planificar un *kernel* de mayor prioridad cuando un *kernel* de menor prioridad ha finalizado. Este esquema puede tener un tiempo de respuesta lento cuando se ejecutan *kernels* largos. Algunos autores también han empleado heurísticas para establecer el mejor orden de ejecución de un conjunto de *kernels*. Estas heurísticas necesitan conocer el tiempo de ejecución de los *kernels* con acceso exclusivo a la GPU, y coejecuta *kernels* con distintos número de CTAs con CTAs *dummy*. Así, este esquema propuesto para la coejecución de *kernels* no es flexible y no puede aplicarse sobre la marcha porque el *profiling* necesario es muy costoso.

Otros autores han propuesto la construcción de *macrokernels* para ejecutar *kernels* de forma concurrente. Liang Y. et al. [50] usan los *macrokernels* para combinar dos *kernels*, lo que requiere la utilización de matrices de memoria adicionales para guardar el patrón de planificación de los *kernels* combinados y los índices para la identificación de los CTAs e hilos correspondientes a cada *kernel*.

Wu B. et al. [100] han planteado transformaciones *SM-Centric* del código del *kernel* para la coejecución de dos *kernels* en la GPU. Esos *kernels* se asignan a diferentes SMs utilizando un esquema *filling-retreating* o llenado-desalojo. La mejor distribución de SMs se encuentra en una etapa *off-line* que consta de dos fases en las que deben probarse muchas combinaciones para la coejecución. También han utilizado un método de *preemption* que permite desalojar los CTAs de los *kernels* en coejecución, pero no implementa ningún método para reiniciar los CTAs desalojados. En [101] se han centrado en el desarrollo de un mecanismo de *preemption* eficiente. Así, aplicando la *preemption* de forma espacial, han permitido la coejecución de los *kernels*. Sin embargo, el planificador desarrollado sólo considera la ejecución de dos *kernels* con diferentes prioridades. Yu C. et al. [104] han propuesto SMGuard, donde los SMs físicos se dividen conceptualmente en pequeños *slices* donde los CTAs de los *kernels* son ejecutados. Estos *slices* son agrupados en unidades *software* denominadas *CapSM*. De esta forma, un *CapSM* puede agrupar los CTAs que se ejecutan en diferentes SMs. El planificador se basa en la reserva de recursos, es decir, en el número de CTAs que lanzará el *kernel*. En este mecanismo no se tiene en cuenta la interferencia producida por los *kernels* durante la coejecución, lo que dificulta el desarrollo de políticas de

planificación dinámica precisas basadas en el rendimiento o la calidad de servicio (QoS). Además, dada la forma en la que se organizan los *CapSM* (los CTAs de un *CapSM* pueden estar ubicados en diferentes SMs) no se puede desarrollar un modelo de coejecución preciso.

Chen Q. et al. [20] han implementando una política de QoS, sin utilizar un mecanismo de *preemption*, desarrollando un modelo para predecir la degradación del rendimiento de las aplicaciones sensibles a la latencia en los aceleradores debido a la coejecución de las aplicaciones. El modelo requiere un *profiler* exhaustivo *off-line*, lo que restringe la utilización del modelo en situaciones reales.

4.2. Ejecución Concurrente de *Kernels* (CKE)

Tal y como se describió en la sección 2.2, el *hardware* encargado de ejecutar las instrucciones de un *kernel*, en la arquitectura NVIDIA, son los *cores* de los *Streaming Multiprocessors* (SMs). Estos SMs poseen una red de interconexión que sirve de puente entre la memoria caché de cada SM y la memoria principal de la tarjeta. La asignación de un gran número de recursos de cómputo a una aplicación que genera una gran cantidad de peticiones a memoria puede provocar la saturación del canal de memoria. El canal de memoria posee un ancho de banda específico, por lo que si los SMs generan peticiones a una velocidad mayor que la velocidad con la que la memoria puede resolver dichas peticiones, ciertas operaciones se verán penalizadas en cuanto al tiempo de ejecución. Esto se debe a que la petición a memoria realizada por el SM que ejecuta dicha operación se ha visto retrasada por la saturación del canal. Además, la unidad de *load/store* puede colapsarse, por ejemplo al quedarse sin entradas en la *Pending Request Table*, la cual contiene todas las peticiones de datos en curso realizadas a la memoria. Existen otras causas de saturación de recursos, como los bloqueos del canal (*pipeline stalls*) por las dependencias de tipo RAW (en los *kernels* intensivos en cómputo) o el *trashing* de la caché L1 (en los *kernels* intensivos en caché L1) [103].

La liberación de los recursos de cómputo para reducir la velocidad con la que llegan las peticiones al sistema de memoria permite eliminar el cuello de botella en la misma. Esto permite asignar los recursos liberados a otra aplicación para su ejecución simultánea. La aplicación a la que se le asignan los recursos liberados tampoco debería saturar el canal de memoria, ya que provocaría el mismo efecto, produciendo de nuevo un cuello de botella. Si la combinación de recursos de cómputo asignados a cada aplicación es adecuada, se puede maximizar el rendimiento de la GPU reduciendo el tiempo de ejecución necesario para la

ejecución de un conjunto de aplicaciones, o sería posible la aplicación de ciertas políticas de planificación que permitan hacer un reparto justo de los recursos. El reparto de los recursos de cómputo entre un conjunto de aplicaciones que van a ser coejecutadas en la GPU es una tarea esencial que marcará el rendimiento máximo obtenido.

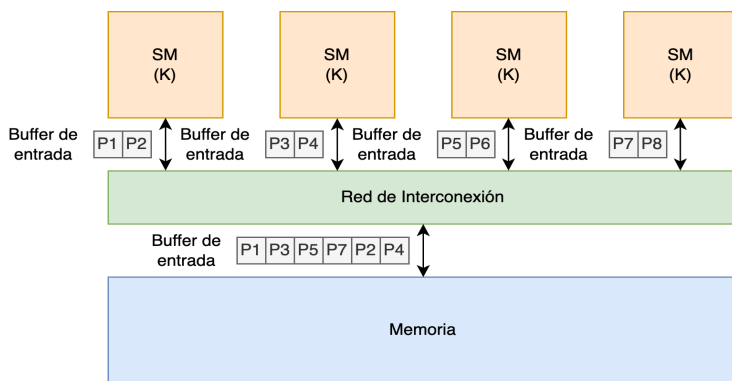


Figura 4.1: Distribución simple de SMs. Un único *kernel* (K) es lanzado en la GPU, por lo que sus bloques se distribuyen entre todos los SMs. En cada SM, los bloques del *kernel* realizan hasta dos peticiones (P) por ciclo a la red de interconexión por medio de sus *buffers* de entrada. Finalmente, la red de interconexión selecciona hasta seis peticiones ordenándolas en un *buffer* de entrada a memoria.

En la figura 4.1 se muestra un ejemplo con una GPU que tiene cuatro SMs, donde cada SM es capaz de realizar hasta dos peticiones a memoria (P) al mismo tiempo, mientras que la memoria es capaz de servir hasta seis peticiones. El número de peticiones realizadas depende de las características del *kernel*. En este ejemplo, se lanza un único *kernel* (K), por lo que sus bloques ocupan todos los SMs de la GPU. El *kernel* K realiza, por cada ciclo de ejecución, dos peticiones a memoria por cada SM. La memoria, por su parte, es capaz de servir hasta seis peticiones por cada ciclo de ejecución. Dado que el número de peticiones realizadas en cada ciclo supera el número de peticiones que puede servir la memoria, se produce un cuello de botella que bloquea la ejecución de los bloques de hilos que han realizado las peticiones de memoria que no han podido servirse, y los recursos de cómputo pueden quedar ociosos. En consecuencia, las peticiones futuras y la entrada de nuevos *warps* o bloques quedaría retrasada haciendo que el rendimiento no aumente o incluso disminuya con el aumento de *cores* asignados.

Algunas soluciones a este problema, las cuales consisten en coejecutar varios *kernels*, se plasman en las figuras 4.2 y 4.3. En la figura 4.2 se aprecia el resultado

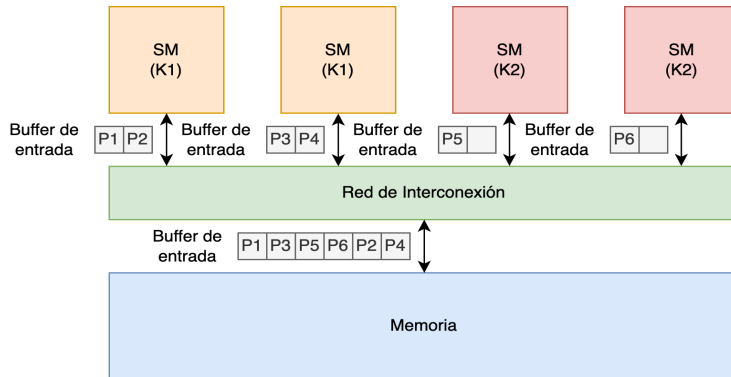


Figura 4.2: Distribución SMT de SMs. En esta distribución sobre los cuatro SMs de una GPU, se asignan los bloques del *kernel* 1 (K1) a dos de los SMs, mientras que los bloques del *kernel* 2 (K2) se asignan a los otros dos SMs. Los bloques de K1 realizan dos peticiones por ciclo, mientras que los de K2 realizan solo una. Dado que el *buffer* de entrada a memoria puede servir hasta seis peticiones al mismo tiempo, todas las peticiones de los SMs pueden ser servidas.

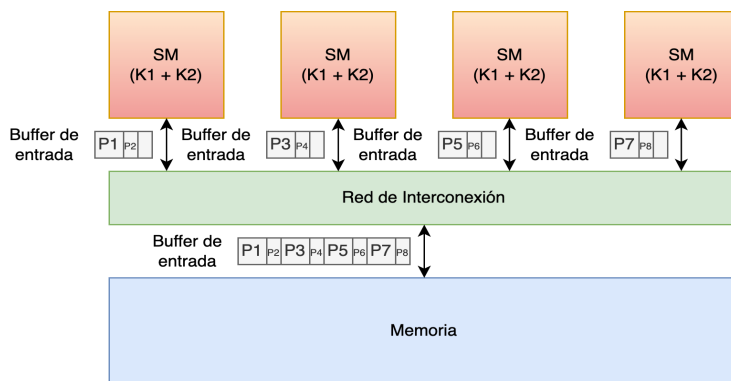


Figura 4.3: Distribución SMK de SM. En esta distribución sobre los cuatro SMs de una GPU, se realiza la misma distribución de bloques de los dos *kernels* K1 y K2 en cada uno de los SMs. Suponiendo que los *kernels* se reparten sobre cada SM a partes iguales, K1 haría una única petición por ciclo, mientras que K2 haría el equivalente a la mitad de una petición (es decir, una petición cada dos ciclos). Esto permite reducir la carga sobre la memoria, por lo que la red de interconexión es capaz de atender las cuatro peticiones de K1 y las dos peticiones ($1/2$ peticiones \times 4 SMs) de K2.

obtenido al aplicar una distribución *spatial multitasking* (SMT). En esta distribución se reduce el número de SMs a los que se les asigna el *kernel* intensivo en memoria (K1) a dos, por lo que los otros dos SMs quedan libres para ser usados por un nuevo *kernel* (K2). K1 sigue realizando dos peticiones a memoria, mientras que K2 realiza solo una. Esta reducción de peticiones permite reducir la carga al sistema de memoria, por lo que se pueden responder a todas las peticiones realizadas por ambos *kernels* sin que los bloques queden bloqueados y, de esta forma, puedan continuar con su ejecución. Otra posibilidad consiste en aplicar la distribución *simultaneous multikernel* (SMK), reflejada en la figura 4.3, en la que se puede apreciar algo similar. En este caso los bloques de los dos *kernels* comparten los *cores* de cada SM a partes iguales, por lo que las peticiones de K1 en cada SM se reducen a una única petición, mientras que K2 realiza la mitad de peticiones. Esto permite servir todas las peticiones realizadas, ya que la memoria puede servir hasta seis peticiones completas. Cuatro de ellas pertenecen a K1 y las otras dos a las cuatro mitades de K2. La coejecución de dos aplicaciones en una GPU puede mejorar el rendimiento del sistema, ya que permite optimizar el uso de los recursos de cómputo y de memoria.

4.3. Motivación

Los esquemas descritos en la sección 4.2 han sido probados con la ejecución de dos aplicaciones en una GPU TITAN X de NVIDIA con arquitectura Pascal y 28 *cores*. Una de las aplicaciones puede catalogarse como intensiva en memoria (VA) y la otra como intensiva en cómputo (PF). Los resultados de las coejecuciones se muestran en la tabla 4.1. Esta tabla muestra la coejecución de las aplicaciones VA y PF con las distribuciones SMT y SMK. En la distribución SMT se ha usado la mejor combinación posible que consiste en una distribución de 19 SMs para VA y 9 para PF, consiguiendo que la velocidad de ejecución sea 1,44 veces más rápida que la ejecución secuencial. Por otro lado, para la distribución SMK se ha comprobado que la mejor combinación de bloques dentro de un SM para esta misma pareja es de 7 bloques para VA y 1 para PF. Esta distribución ha hecho que la velocidad de ejecución de las dos aplicaciones sea 1,37 veces más rápida que su ejecución secuencial.

	SMT		SMK	
(VA/PF)	SMs	Mejora	Bloques	Mejora
(VA/PF)	19/9	1,44	7/1	1,37

Tabla 4.1: Mejora de rendimiento SMT y SMK.

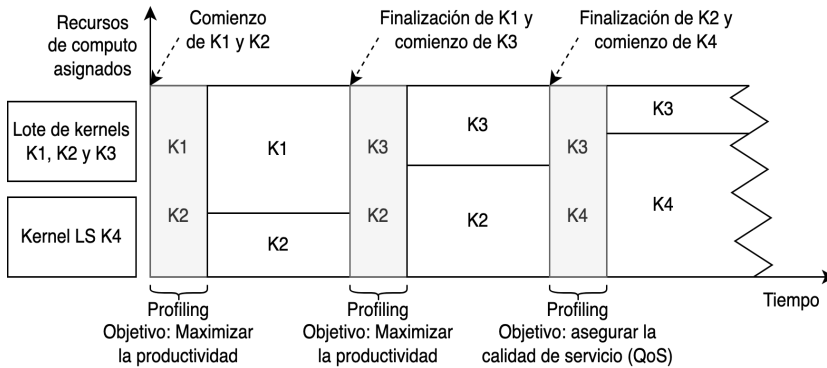


Figura 4.4: Línea de temporal que muestra una visión general del enfoque *software* propuesto para la coejecución de *kernels*. Cuando se planifica un nuevo *kernel*, se inicia una fase de *profiling* productivo para establecer la mejor partición dentro de un SM (cajas grises). Como puede verse, el método se aplica sobre la marcha a un número arbitrario de *kernels* nuevos que llegan con diferentes objetivos de planificación.

La figura 4.4 muestra un ejemplo motivador de nuestra propuesta, en el que la coejecución de los *kernels* es explotada por un planificador que implementa políticas orientadas tanto al alto rendimiento como a la baja latencia. La figura 4.4 muestra la planificación temporal de cuatro *kernels* y la distribución relativa de los recursos de la GPU entre ellos, es decir, el número de CTAs asignados a cada *kernel*. Tres de ellos son *kernels* que pertenecen a un *batch*, es decir, suponemos que tienen la misma prioridad y el objetivo es coejecutarlos lo más rápido posible para obtener un alto rendimiento global, y el restante es un *kernel* LS (*Latency Sensitive*) que debe ejecutarse con algunas restricciones temporales específicas con un tiempo de respuesta. Inicialmente, sólo los *kernels* del *batch* están listos para ejecutarse. El planificador selecciona los *kernels* K1 y K2 para su ejecución conjunta y trata de maximizar el rendimiento de la ejecución conjunta para lograr una tasa de ejecución de los *kernels* elevada. El planificador podría disponer de alguna información básica sobre el *kernel*, por ejemplo, si los *kernels* son intensivos en memoria o en cómputo, para ayudarlo a elegir un par de *kernels*, pero también podría llevarse a cabo una selección aleatoria. En cualquier caso, el planificador inicia una fase de *profiling* para obtener una buena partición de los recursos de la GPU entre los dos *kernels*. Esta fase se muestra con un caja gris a partir de un tiempo t_1 . Durante esta fase, se evalúa el rendimiento de la coejecución y, si es necesario, se modifica la distribución de los recursos. Obsérvese que no se representa ninguna asignación específica de CTAs durante esta fase porque varía

con el tiempo. Una vez se encuentra una buena distribución, la fase de *profiling* finaliza y los *kernels* continúan su ejecución hasta que uno de ellos termina. La altura de la caja de cada *kernel* indica la proporción relativa de recursos asignados. En el momento t_2 , $K1$ termina y el planificador selecciona y lanza $K3$. Una vez más, se inicia una fase de *profiling* para encontrar una buena partición de recursos entre $K2$ y $K3$. A continuación, $K2$ y $K3$ se ejecutan hasta que, en el tiempo t_3 , un *kernel* LS, $K4$, está listo. Debido a este evento, el planificador desaloja uno de los *kernels* que están ejecutándose, $K2$ en este ejemplo, y lanza $K4$. Se inicia una nueva fase de *profiling*, pero ahora el objetivo del planificador es cumplir los requisitos de QoS para $K4$ y, como efecto secundario, se asignan menos recursos a $K3$. Una vez que se encuentra una asignación adecuada, la fase de *profiling* termina y los *kernels* siguen coejecutándose hasta el siguiente evento.

4.4. FlexSched

El enfoque *software* que presentamos en esta tesis se denomina *FlexSched*. Este sistema se ejecuta en el *host*, y se encarga de planificar un conjunto de aplicaciones de la GPU aplicando criterios orientados tanto al rendimiento como a la calidad del servicio (QoS). A partir de modificaciones simples en el código fuente del *kernel*, desarrolla mecanismos flexibles para ejecutar *kernels* concurrentemente que incluyen el desalojo parcial del *kernel* y su lanzamiento. Además, se implementa un método de *profiling* productivo *on-line* para obtener información de los *kernels* coejecutados en tiempo de ejecución. De este modo, el planificador puede establecer una partición de recursos adecuada para los *kernels* concurrentes.

En este modelo vamos a denominar subtarea al trabajo que realiza cada uno de los CTAs. Para proporcionar un esquema CKE flexible, el *grid* del *kernel* original se transforma en un *grid* nuevo formado por CTAs persistentes [100]. De esta forma, cada CTA persistente se encargará del trabajo que realizaban varios CTAs en el *grid* original, es decir, de varias subtareas. Además, estos CTAs persistentes se agrupan en unidades que denominamos BSU (*Basic Scheduling Units*), y se corresponden con las entidades de planificación más pequeñas gestionadas por nuestro planificador. El uso de las BSUs permite controlar la distribución precisa de CTAs en los SMs durante el *profiling* y evita la costosa operación de desalojo y relanzamiento del *kernel*. La figura 4.5 muestra un esquema de este enfoque. Se puede observar que varias BSUs de dos *kernels* se ejecutan en la GPU, mientras que los módulos *rtSMK*, Planificador y *Profiler* se ejecutan en la CPU. Las principales características de estos módulos son las siguientes:

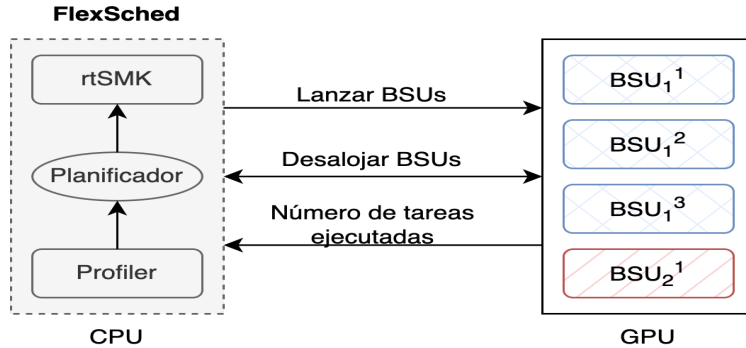


Figura 4.5: Visión general de FlexSched. Consta de un lanzador de BSUs (*rtSMK*), un planificador y un *Profiler*, todos ellos ejecutados en la CPU o *host*. En este ejemplo se lanzan tres BSUs (BSU_1^1 , BSU_1^2 y BSU_1^3) del *kernel* 1 y un BSU (BSU_2^1) del *kernel* 2, que se ejecutan en la GPU. Las flechas que conectan la CPU con la GPU representan los comandos de lanzamiento y desalojo iniciados por *rtSMK* y el planificador, respectivamente, mientras que los comandos de transferencia de datos son gestionados por el *profiler* para obtener al instante el número de subtareas ejecutadas por cada *kernel*. El *profiler* envía esta información al planificador, que evalúa si la configuración actual de las BSUs es adecuada.

- *Profiler*: este módulo es responsable, durante la fase de *profiling*, de obtener una estimación del trabajo completado por cada *kernel* en estado *running*. Para ello el módulo está continuamente sondeando, a través del bus que conecta el *host* con el dispositivo, cuantas subtareas se han completado para de esa forma calcular la tasa de ejecución de subtareas instantánea. Las BSUs pertenecientes al mismo *kernel* colaboran para incrementar el número de subtareas ejecutadas de dicho *kernel*. A continuación, el valor calculado se envía al planificador para que se produzcan cambios en la asignación de las BSUs actuales, si fuera necesario.
- *Planificador*: utiliza la información proporcionada por el *Profiler* y el modelo de rendimiento, que puede ser diferente en función del objetivo de la planificación, para dar instrucciones al módulo *rtSMK* para que lance las BSUs pertenecientes a los *kernels* coejecutados. Además, supervisa su progreso y, en caso de desviación con respecto a la predicción del modelo, toma una decisión sobre la reasignación de CTAs. Cuando finaliza un *kernel* durante la coejecución, el planificador también selecciona el nuevo

kernel a lanzar y realiza la reasignación de recursos si fuera necesario. La reasignación se realiza desalojando y relanzando las BSUs de los distintos *kernels*. Así, cualquier *kernel* en estado *running* siempre está progresando, excepto cuando se desalojan todas las BSUs de dicho *kernel*.

- *rtSMK*: se encarga de organizar la computación del *kernel* con BSUs y de lanzar las BSUs según las órdenes del planificador.

4.5. Detalles de la implementación

En esta sección se detallan los cuatro elementos clave de la implementación de *FlexSched*. En primer lugar, se muestran las modificaciones que deben aplicarse al código fuente del *kernel* antes de que pueda ser utilizado por nuestro planificador. A continuación, se introduce el mecanismo utilizado para asignar CTAs a los SM, es decir, *rtSMK*. Posteriormente, se discute el conjunto de posibles configuraciones de asignación de CTAs para dos kernels coejecutados y, finalmente, se presenta la métrica utilizada durante la fase de *profiling*.

4.5.1. Transformación del Kernel

La implementación del mecanismo de *preemption* o desalojo requiere una transformación de código fuente del *kernel*, pero esta modificación puede automatizarse fácilmente como se explica en [20, 101]. En primer lugar, se modifica el *grid* del *kernel* original para ejecutar el *kernel* utilizando CTAs persistentes. Así, nuestro esquema puede lanzar un número de CTAs igual o inferior a $max(CTA_{SM}) \cdot numSMs$, donde $max(CTA_{SM})$ es el número máximo de CTAs que pueden asignarse simultáneamente en un SM y $numSMs$ es el número total de SMs en la GPU. Esta transformación puede ser realizada automáticamente por un compilador, como se ha demostrado en [19].

La transformación propuesta tiene dos ventajas a la hora de implementar el mecanismo de *preemption*. Por un lado, los CTAs persistentes suelen ejecutar varias iteraciones en las que, en cada iteración, se computa el trabajo de un solo CTA del *grid* original. Cada iteración podemos asemejarla al trabajo necesario para completar una subtarea. Así, el desalojo puede tener lugar al final de cada iteración (subtarea) y, cuando se relanza un *kernel* previamente desalojado, cada CTA sólo reanuda la ejecución desde la última iteración o subtarea completada. Por otra parte, el mecanismo de *preemption* funciona más rápidamente ya que, para los tamaños de *grid* más comunes, sólo están activas decenas (como mucho

unos cientos en una arquitectura moderna) de CTAs persistentes en lugar de varios miles.

También se ha desarrollado un mecanismo flexible para la distribución de la carga entre los CTAs. En lugar de asignar una computación específica a cada CTA con un mapeo fijo (como en [50]), los CTAs obtienen la carga de trabajo actualizando atómicamente una variable común en memoria global al comienzo de cada iteración, como también se hace en [100]. Esta variable de memoria global actúa como un contador (que parte de cero) que incrementa un índice de subtareas y desempeña un papel importante durante el *profiling online*. Este ordenamiento de la ejecución de las subtareas no afecta a la ejecución original del *kernel* y beneficia al mecanismo de *preemption* porque sólo debe guardarse el índice de la última subtarea ejecutada cuando se desaloja un *kernel*.

```

1  /***** GPU code *****/
2  Kernel_func(list_of_original_params, int MaxNumTask, int *State, int
   *TaskCont) {
3      while(true) {
4          // Chequeo de estado (solo un thread)
5          if (threadIdx.x == 0) {
6              if (*State == EVICTED)
7                  blockIdx = -1;
8              else
9                  blockIdx = atomicAdd(TaskCont, 1);
10         }
11         // Sincronizacion del bloque de threads
12         _sync_threads();
13         // Chequeo de finalizacion: no hay mas tareas o es desalojado
14         if (blockIdx >= MaxNumTask || blockIdx == -1)
15             return;
16         // El codigo original del kernel va aqui
17     }
18 }
19 /*****Codigo de llamada de la CPU *****/
20 Kernel_func<persist_grid_size> (list_of_original_params, MaxNumTask,
   State, TaskCont);

```

Código 4.1: Modificaciones en el *kernel* original para el mecanismo de *preemption*.

En el código 4.1 se indican los cambios clave del *kernel*. Como puede observarse, se requieren dos nuevas variables en memoria global para cada *kernel* (línea 2). Una de estas variables, denominada *State*, guarda el estado del *kernel* en la GPU y puede tomar tres valores posibles: *Ready*, *Running* o *Evicted*. Cuando un *kernel* ha resuelto todas sus dependencias de datos, y sus datos de entrada han sido transferidos desde el *host* al dispositivo, toma el estado *Ready*. Justo antes de que se lance un *kernel*, se pone en el estado *Running*. La otra variable en memoria, denominada *TaskCont*, contiene el índice de la siguiente

subtarea disponible (inicialmente fijado en 0). Además, hay un nuevo parámetro, *MaxNumTask* (también en la línea 2), que contiene el número total de subtareas a ejecutar y que es comprobado por los hilos de la GPU para terminar cuando todas las subtareas han sido realizadas.

El código 4.1 también muestra la inclusión del mecanismo de desalojo y la estrategia de distribución de subtareas. El cálculo original que realizaban los CTAs está encerrado en un bucle *while* (línea 3) que termina cuando el planificador en la CPU envía una orden de desalojo o cuando no hay más subtareas pendientes (línea 14). Al principio de cada iteración del bucle *while*, el hilo del CTA con identificador 0 lee la variable *State* (línea 6). Si el estado ha cambiado a *Evicted*, entonces una variable mapeada en la memoria compartida, llamada *blockId*, se pone a -1 (línea 7). Esta variable también se utiliza para almacenar el índice de la subtarea cuando el *kernel* se está ejecutando (línea 9). Como puede observarse también en la línea 9, los nuevos valores del índice de la subtarea son obtenidos por el hilo 0 en cada iteración del CTA ejecutando una instrucción *atomicAdd* en la variable *TaskCont* mientras que los restantes hilos del bloque esperan en una instrucción de sincronización del bloque (línea 12). Después de esta barrera, todos los hilos del bloque leen el *blockId* y comprueban (línea 14) la condición de terminación (todas las subtareas del *kernel* han sido computadas o el contenido del estado ha cambiado a *Evicted*). Si la condición no se cumple, el CTA computa una nueva subtarea con el valor actual de *blockId*. También hay que aplicar una modificación adicional al código original para cambiar la indexación empleada durante la computación. En el código fuente de los *kernels*, esta indexación se implementa normalmente utilizando el índice del CTA, mientras que ahora debe utilizarse el valor de *blockId*. Estas transformaciones pueden aplicarse automáticamente como se muestra en [20, 101].

4.5.2. Basic Scheduling Units (BSUs)

rtSMK es el módulo de *FlexSched* encargado de planificar los *kernels* para que se ejecuten de forma concurrente utilizando un mecanismo basado en *software* para distribuir los recursos disponibles de la GPU. Dado que nuestro planificador se ve obligado a emplear el planificador *hardware* de CTAs, aprovechamos la política de asignación de CTAs de tipo *round-robin* [50] para implementar un esquema de asignación de CTA *intra-SM*, en el que CTAs de diferentes *kernels* se asignan dentro del mismo SM. Así, nuestro enfoque define la *Basic Scheduling Unit* (BSU) como el conjunto de CTAs agrupados por el número de SM que hay en la GPU. De este modo, cuando se lanza una BSU en una GPU, se asignará un CTA a cada SM. Si se lanza otra BSU, se ejecutará un nuevo CTA en cada SM



siempre que haya suficientes recursos para ellos. Como resultado, dos CTAs, que pertenecen a diferentes BSUs, se estarán ejecutando en cada SM. El uso de las BSU conduce a un esquema de distribución de CTA similar a SMK [45, 98], que explota la heterogeneidad de los *kernels* permitiendo una distribución de grano fino por parte de múltiples *kernels* dentro de cada SM.

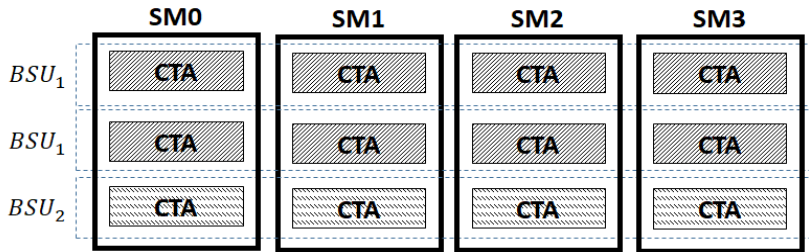


Figura 4.6: Planificación de tres BSUs en una GPU con cuatro SMs. Cuando se carga una BSU, un CTA es lanzado en cada SM como se indica en los rectángulos delimitados con líneas discontinuas (BSU_1 y BSU_2). BSU_1 y BSU_2 identifican a las BSUs pertenecientes a los *kernels* $k1$ y $k2$ respectivamente. Así, se ejecutan dos BSUs de $k1$ y una BSU de $k2$.

La figura 4.6 muestra un ejemplo en el que tres BSUs pertenecientes a dos *kernels* se planifican en una GPU con cuatro SMs. En esta figura, la BSU_k corresponde a una BSU del *kernel* k . Las BSUs que pertenecen al mismo *kernel* colaboran entre si para realizar la computación del *kernel*, ya que comparten el contador $TasksId$ (véase el código 4.1). Este es el caso de las dos instancias de la BSU_1 en la figura 4.6. Sin embargo, pueden ser desalojadas independientemente porque cada BSU tiene su propia variable *State*. Por lo tanto, se puede llevar a cabo una asignación dinámica de CTAs eficiente y receptiva, ya que una BSU desalojada puede dejar espacio para las BSUs de otro *kernel*, siempre que las nuevas BSUs puedan ejecutarse con los recursos liberados por la BSU desalojada.

La granularidad de una BSU se define como el número de CTAs que se lanzarán en cada SM. Así, una BSU con un valor de granularidad de uno, como las representadas en la figura 4.6, lanzará $numSMs$ CTAs, donde $numSMs$ es el número de SMs del dispositivo. No obstante, la granularidad puede ser superior a uno para reducir la sobrecarga de la gestión de las BSUs en los *kernels* con un elevado número de CTAs persistentes por SM. Así, la granularidad de una BSU puede oscilar entre 1 y el número máximo de CTAs persistentes por SM. Nuestro planificador es capaz de gestionar las BSUs con diferente granularidad atendiendo a las decisiones en tiempo de ejecución.

4.5.3. Configuración del espacio de coejecución

Uno de los principales problemas que hay que resolver para obtener beneficios de la ejecución concurrente es establecer una buena distribución de los recursos de la GPU para que los *kernels* se puedan ejecutar concurrentemente. Como resultado de esta distribución, las CTAs pertenecientes a diferentes *kernels* se coejecutan en la GPU. Nuestro enfoque de asignación de CTAs sigue un esquema SMK, por lo que se realiza la coejecución *intra-SM* y todos los SM tienen una distribución de CTAs idéntica. Sin embargo, como los CTAs de una BSU planificada deben ser persistentes, los requisitos de ocupación de recursos deben comprobarse antes de lanzar una nueva BSU para garantizar que los nuevos CTAs puedan asignarse correctamente a los SMs. El número válido de BSUs planificadas de *kernels* diferentes puede calcularse por adelantado conociendo las siguientes métricas relativas a los CTAs de los *kernels* a coejecutar: número de hilos por CTA, número de registros necesarios por hilo y espacio de memoria compartida por CTA. Todos estos valores se conocen durante la compilación del código. Así, utilizando estas métricas y teniendo en cuenta la capacidad de la arquitectura de la GPU, es sencillo establecer una configuración válida para la ejecución concurrente, es decir, el número de BSUs de los *kernel* a coejecutar que se pueden lanzar.

Más concretamente, una configuración válida debe cumplir dos condiciones: todos las CTAs deben ser persistentes y no se pueden asignar más CTAs. Esta última condición garantiza que una configuración válida debe agotar al menos uno de los recursos del SM para que no se puedan asignar más CTAs. Denominamos al conjunto de configuraciones válidas de coejecución como espacio de configuración de coejecución, CCS. Este espacio se compone de un conjunto ordenado de tuplas que indica el número de CTAs que se coejecutan en cada SM para dos *kernels*, como se refleja en la ecuación 4.1, donde K_1 y K_2 son los *kernels* concurrentes y $|BSU_i^j|$ indica el número de BSUs lanzadas del *kernel* K_i en la j -ésima configuración de coejecución. Para facilitar la búsqueda de una buena configuración, el conjunto de configuraciones se ordena por valores crecientes de $|BSU_i^j|$, es decir, $|BSU_i^j|$ toma el valor más bajo de CTAs concurrentes por SM de K_1 para la primera configuración. El número de CTAs aumenta a medida que pasamos a índices de configuración más altos. Así, $|BSU_1^n|$ representa el mayor número de CTAs por SM para el *kernel* K_1 que pueden ejecutarse concurrentemente con los CTAs del *kernel* K_2 . Es fácil deducir que los valores que toma $|BSU_2^j|$ tienen justo el comportamiento contrario, es decir, disminuyen a medida que aumenta el índice de la configuración.

$$CCS(K_1, K_2) = (|BSU_1^1|, |BSU_2^1|), \dots, (|BSU_1^n|, |BSU_2^n|) \quad (4.1)$$

Por ejemplo, supongamos que los recursos que necesita cada *kernel* permiten asignar un máximo de 8 CTAs por SM para cada *kernel*. Sin embargo, cuando se ejecutan de forma concurrente, los recursos del SM deben compartirse, lo que da lugar a siete configuraciones posibles, $CCS = \{(1, 7), (2, 6), (3, 5), (4, 4), (5, 3), (6, 2), (7, 1)\}$. Así, en la primera configuración una BSU del primer *kernel* y siete BSUs del segundo *kernel* pueden ejecutarse concurrentemente.

4.5.4. Profiling de la coejecución y planificación

El cuarto elemento clave en la implementación de *FlexSched* es la fase de *profiling*. Durante esta fase se monitoriza el CCS de dos *kernels* para encontrar la mejor configuración de coejecución atendiendo a alguna política de planificación específica que tenga en cuenta el progreso de cada *kernel*, por ejemplo la optimización del *system throughput*. Para ello se podría considerar el progreso normalizado tal y como fue definido en la ecuación 2.1, pero entonces deberíamos conocer el valor de *IPC* lo cual supone una costosa fase de *profiling* usando la librería CUPTI. Una alternativa más ligera, de grano grueso, consiste en calcular el número de tareas completadas por unidad de tiempo. Concretamente, nuestro *profiler* puede determinar la tasa de ejecución de subtareas, *TER*, alcanzada por las BSUs en ejecución de un *kernel*. El valor TER_i^j del *kernel* i con la configuración j del CCS se calcula mediante la ecuación 4.2, donde $TaskCont_i$ es el contador de subtareas compartido por todas las BSUs del *kernel* K_i lanzadas para la configuración j , y T_i es el tiempo transcurrido desde que el *kernel* comenzó su ejecución.

$$TER_i^j = \frac{TaskCont_i}{T_i} \quad (4.2)$$

Este valor de *TER* es una aproximación de grano grueso al valor de *IPC*, por lo que podríamos usarlo en su lugar dentro de la ecuación 2.1 que define el progreso normalizado de un *kernel*. Para ello deberíamos calcular el valor de *TER* cuando el *kernel* se ejecuta en solitario, pero uno de los retos que nos marcamos con *FlexSched* es que se pueda usar en tiempo real sin recurrir a ejecuciones previas. Por tanto, en lugar de calcular el progreso normalizado comparando el rendimiento del *kernel* coejecutado con su rendimiento en solitario, vamos a calcularlo comparando el rendimiento alcanzado por dos configuraciones j y k del CCS tal y como se muestra en la ecuación siguiente:

$$NP_i^{j,k} = \frac{TER_i^j}{TER_i^k} \quad (4.3)$$

Utilizando el valor de $NP_i^{j,k}$, el planificador puede aplicar una política orientada a, por ejemplo, maximizar el *system throughput*, calculado mediante la ecuación 2.2, pero sustituyendo los valores de NP_i por $NP_i^{j,k}$:

$$STP_S = \frac{1}{2} \cdot (NP_1^{j,k} + NP_2^{j,k}) \quad (4.4)$$

El valor de STP_S se corresponde con el promedio de valores de *speedup* de los *kernels* 1 y 2 de la configuración j con respecto a la configuración k . De esta forma, si el valor es superior a uno, la configuración j consigue un mejor *TER* global que la configuración k . De forma similar, si $STP_S < 1$, la configuración k obtiene una mejor tasa de ejecución de subtareas que la configuración j . Así, se pueden comparar diferentes configuraciones de coejecución de un CCS para encontrar la mejor utilizando un enfoque *hill climbing*.

En conclusión, nuestra propuesta utiliza un *profiling* productivo *on-line* para encontrar la mejor configuración de coejecución en lugar de los esquemas anteriores CKE basados en *software*, que se basan en modelos de coejecución que sólo tienen en cuenta la partición de la GPU para la coejecución de los *kernels* [76, 104]. Además, estos trabajos ignoran la interferencia entre los *kernels* que se produce durante la coejecución, por lo que no son capaces de establecer un buen mapeo de CTAs que cumpla con los criterios de optimización del rendimiento o con las restricciones en el tiempo de ejecución.

4.6. Resultados experimentales

La validación del modelo *software* se ha realizado utilizando diferentes aplicaciones que contienen uno o varios *kernels* pertenecientes a las *suites* de referencia CUDA SDK [66], Rodinia [18] y Chai [27]. Con estas aplicaciones, perseguimos construir una carga de trabajo real en la que se puedan coejecutar *kernels* intensivos en cómputo y en memoria. Estos experimentos se han realizado en un servidor con dos CPUs Xeon(R) E5-2620 y una Nvidia Titan X Pascal conectados por un bus de interconexión PCIe 3.0.

La tabla 4.2 muestra la lista de aplicaciones que hemos utilizado. La mayoría de las aplicaciones tienen un solo *kernel*, pero dos de ellas, *Separable Convolution* y *Canny*, están compuestas por dos y cuatro *kernels*, respectivamente. Los *kernels* de ambas aplicaciones se ejecutan en forma de *pipeline*, ya que la salida de un *kernel* es la entrada del siguiente.

En trabajos previos [45, 109] se ha demostrado que la coejecución de *kernels*

Kernel	Aplicación	Tiempo de ejecución (ms)	Categoría
HST256	Histogram	4,18	MB
BS	Black Scholes	4,47	MB
VA	Vector Addition	7,24	MB
SPMV	Sparse MV Mult.	10,60	MB
RED	Reduction	5,05	MB
CCONV	Separable	1,60	MB
RCONV	Convolution	1,45	MB
GCEDD	Canny	5,26	CB
SCEDD		9,65	CB
NCEDD		7,10	CB
HCEDD		2,15	CB
PF	Path Finder	6,77	CB
MM	Matrix Mult.	10,38	CB

Tabla 4.2: Aplicaciones usadas en la validación del modelo *software*. La mayoría de las aplicaciones tienen un solo *kernel*, menos *Separable Convolution* y *Canny* que están compuestas por dos y cuatro *kernels*, respectivamente. La tercera columna muestra el tiempo de ejecución de cada *kernel*. La cuarta columna indica el recurso más usado por cada *kernel* y la categoría a la que pertenece.

obtiene buenos resultado de rendimiento cuando los *kernels* coejcutados hacen uso de recursos complementarios. De este modo, si hay varios *kernels* listos para ser lanzados, es aconsejable realizar un estudio para encontrar los emparejamientos de *kernels* adecuados. Algunos autores [16, 85] han propuesto enfoques diferentes para la clasificación de los *kernels*. En este trabajo hemos usado la métrica *Kernel Mix Intensity* (KMI) para representar la intensidad operativa de un *kernel* en la GPU. Este valor se obtiene dividiendo el número de instrucciones de cálculo y de memoria ejecutadas por el *kernel*. Atendiendo al valor de KMI resultante, un *kernel* se clasifica en intensivo en cómputo (CB) o en memoria (MB). Para realizar este análisis se ha empleado la librería CUPTI [57] para obtener el número de instrucciones de cada tipo. Concretamente, se ha utilizado la API de métrica de esta librería para calcular el número de instrucciones enteras y en coma flotante (de precisión media, simple y doble). Además, también se ha obtenido el número de instrucciones de memoria DRAM de lectura y escritura. La expresión final de KMI viene dada por la ecuación 4.5.

$$KMI = \frac{\#integer + \#half_float + \#single_float + \#double_float}{\#dram_read_trans + \#dram_write_trans} \quad (4.5)$$

La categoría a la que pertenece cada *kernel* de las aplicaciones seleccionadas, se muestra en la columna más a la derecha de la tabla 4.2.

4.6.1. Coste la transformación del *kernel*

La implementación de *FlexSched* que proporciona un mecanismo para la coejecución de *kernels* basada en BSU con soporte para *preemption* basado en *software*, requiere de la transformación del código fuente original del *kernel* como se explica en la sección 4.5.1. Esta transformación puede provocar un aumento en el tiempo de ejecución del *kernel*, produciendo un coste que podemos cuantificar con el siguiente experimento. Sean T_t y T_o el tiempo total de ejecución de los *kernels* transformados y original, respectivamente. Entonces, el coste en el que incurre esta transformación puede calcularse mediante la ecuación 4.6. La segunda columna de la tabla 4.3 muestra el coste de esta transformación (O_t) para los 13 *kernels*.

$$O_t = \frac{T_t}{T_o} \quad (4.6)$$

Kernel	Coste de la transformación	Delay en la <i>preemption</i> (μs)
HST256	0,91	94
BS	1,00	83
VA	1,00	95
SPMV	1,06	70
RED	0,90	32
CCONV	1,00	46
RCONV	1,01	53
GCEDD	1,05	45
SCEDD	1,05	40
NCEDD	0,97	38
HCEDD	1,04	37
PF	1,00	60
MM	0,96	86

Tabla 4.3: Análisis de la transformación de los *kernels* para el modelo *software*. La columna dos indica el coste incurrido por la modificación del *kernel* respecto al *kernel* original. La tercera columna muestra el retardo en el desalajo (en microsegundos) del mecanismo de *preemption* para cada *kernel*, asumiendo que la transferencia de datos no ocurre concurrentemente con la ejecución del *kernel*.

Atendiendo a la expresión del coste, se esperan valores superiores a uno, ya que la transformación del *kernel*, como se explica en la sección 4.5.1, aumenta el número de instrucciones ejecutadas por cada *kernel*. Sin embargo, hay casos en los que los valores son inferiores a uno. Estos resultados pueden explicarse por el hecho de que la transformación del *kernel* también implica una modificación en el número y la granularidad de los CTAs. Por ejemplo, el *kernel* original de RED realiza dos operaciones diferentes. En primer lugar, cada hilo del CTA acumula el contenido de los datos leídos de la memoria global y, a continuación, se realiza una reducción a nivel de CTA. Como el *kernel* transformado tiene un número menor de CTAs, la reducción requiere menos tiempo. El valor del coste de HST256 puede explicar de forma similar, ya que también hay una reducción final. Centrándonos en los *kernels* con un coste superior a uno, podemos ver que el tiempo de ejecución aumenta, como mucho, un 5%. El valor medio de coste es 0,99, lo que indica que, globalmente, el impacto de la transformación sobre el tiempo de ejecución es insignificante.

4.6.2. Retardo en el desalojo

Un aspecto clave de cualquier mecanismo de *preemption* es su capacidad de respuesta a las órdenes de desalojo. En nuestra implementación, el planificador desaloja un *kernel* escribiendo asíncronamente en la variable *State* de la BSU. Esta variable se asigna en la memoria global de la GPU como se explicó en la sección 4.5.1. Cualquier CTA de la BSU lee esta variable antes de la ejecución de una nueva subtarea y, atendiendo a su contenido, computa toda la subtarea o termina. En consecuencia, se produce un retardo entre la llegada de la orden de desalojo y la finalización del *kernel* en ejecución. Este retardo está directamente relacionado con la frecuencia con la que el CTA lee la variable **State**. Así, una reducción del retardo en el desalojo requiere de lecturas de memoria más frecuentes que, sin embargo, aumentan el coste de la operación atómica para actualizar la variable **TaskCont**. Como nos interesa mantener la modificación del código lo más sencilla posible, la granularidad mínima viene dada por el cómputo de un CTA del *kernel* original (véase código 4.1). No obstante, esta granularidad podría aumentarse si fuera necesario aplicando una técnica de *coarsening* al código CTA.

Hemos medido el retardo en el desalojo de cada *kernel* como en [19]. Las órdenes de desalojo se envían en diferentes momentos de la ejecución del *kernel*, y se mide el tiempo transcurrido hasta que el *kernel* se detiene. Este experimento se ha repetido 15 veces, y se han calculado los valores medios. La tercera columna de la tabla 4.3 muestra el retardo medio de desalojo de cada *kernel*. Todos los

kernels tienen un retardo de desalojo inferior a $100 \mu s$, y el retardo medio de desalojo de todos los *kernels* está en torno a $60 \mu s$. Este valor permite desarrollar políticas de planificación que requieren intervalos de desalojo cortos.

4.6.3. Rendimiento de la planificación de CTAs

En la figura 4.5 se muestran los tres módulos que componen *FlexSched*. En esta sección nos interesa evaluar el rendimiento alcanzado por uno de estos módulos, *rtSMK*, que se encarga de la asignación de CTAs en los SMs siguiendo una planificación SMK. Dado que *rtSMK* es el mecanismo utilizado por nuestro enfoque *software* para lanzar CTAs pertenecientes a diferentes *kernels* en los SMs de la GPU, puede compararse con otro método con una funcionalidad similar como cCUDA [85]. cCUDA presenta un mecanismo para la coejecución de *kernels* basado en la fragmentación del *kernel* [109]. Emplean dos *streams* para lanzar múltiples *slices* de ambos *kernels* a coejecutar. De este modo, los *slices* de ambos *kernels* pueden ejecutarse de forma concurrente siempre que ningún *slice* agote todos los recursos del SM. Esta técnica también requiere la modificación de los *kernels*, ya que debe aplicarse la rectificación del índice del CTA para garantizar el comportamiento correcto del *kernel*. Además, la dimensión original del *grid* del *kernel* debe pasarse como parámetro. El *slice* del *kernel* incurre en un coste causado por el lanzamiento múltiple de *slices* del *kernel*. Este coste puede reducirse aumentando el tamaño del CTA.

Para comparar el rendimiento vamos a usar el valor de *STP* alcanzado, pero en lugar de usar el *IPC* para calcular el progreso de cada *kernel* vamos a usar su tiempo de ejecución. Dado que el tiempo de ejecución de cada *kernel* durante una coejecución es diferente, el tiempo de ejecución concurrente (T^{shared}) considera únicamente el rendimiento cuando ambos *kernels* se ejecutan simultáneamente. Para ello *rtSMK* anota el número de subtareas ejecutadas por el *kernel* de mayor duración cuando termina el *kernel* de menor duración. Entonces, el tiempo de ejecución secuencial (T^{alone}) se calcula lanzando secuencialmente ese par de *kernels* utilizando el mismo número de subtareas, es decir, todas las subtareas para el *kernel* más corto y el número de subtareas ejecutadas para el *kernel* más largo. De forma similar, para cCUDA se cuenta el número de CTAs ejecutados por ambos *kernels* cuando están coejecutándose y el tiempo de ejecución secuencial se calcula lanzando todos los CTAs del *kernel* más corto y el número de CTAs anotados para el *kernel* más largo. La ejecución secuencial de los *kernels* se lleva a cabo utilizando diferentes *streams* para cada *kernel*. De este modo, el solapamiento de los *kernels* puede seguir produciéndose cuando está terminando el primero de los *kernels* y por tanto quedan libres suficientes recursos *hardware*

para que se empiece a ejecutar el segundo *kernel*. El *system throughput* alcanzado por los dos *kernels* ($K1$ y $K2$) cuando se ejecutan de forma concurrente viene dado por la ecuación 4.7.

$$STP = \frac{T^{alone}(K1, K2)}{T^{shared}(K1, K2)} \quad (4.7)$$

En la figura 4.7.a, se utiliza una gráfica de barras para representar los mejores valores de *STP* alcanzados por los pares de *kernels* limitados por la memoria y limitados por computación para ambos métodos (*rtSMK* y *cCUDA*). Los valores de **STP** se calculan explorando todos los CCS para cada par de *kernels* utilizando un enfoque de fuerza bruta y se toma la configuración de coejecución (CC) con la mayor puntuación. Para realizar una comparación justa, se emplean los mismos valores para el *coarsening* del CTA en ambos métodos.

Al analizar los resultados mostrados en la figura 4.7.a puede observarse que nuestro esquema de asignación de CTAs propuesto (*rtSMK*) obtiene valores de *STP* que oscilan entre 1,1 y 2,1, lo que confirma las ventajas de la coejecución de los *kernels*. Comparando los valores máximos de *STP* de *rtSMK* y *cCUDA*, podemos ver que la mayoría de las veces nuestro enfoque mejora los resultados de *cCUDA*. Este comportamiento se explica con:

- *cCUDA* incurre en un coste cada vez que se lanza un nuevo *slice*. Así, si el tiempo de ejecución de los *slices* es corto y el número de *slices* lanzados es alto, el coste no es despreciable. Un ejemplo de este comportamiento puede observarse durante la coejecución de RED y HCEDD. El tiempo de coejecución de un *slice* de RED es de solo 12 μs , mientras que el tiempo de lanzamiento del *slice* es de 7 μs . Por tanto, el coste de ejecución de este *kernel* es alto y solo se consigue un *STP* de 0,95. *rtSMK* solo lanza las BSUs una vez, por lo que no se ve afectado por este coste, obteniendo un *STP* de 1,23.
- *cCUDA* también presenta problemas de eficiencia cuando los *slices* terminan. Como los *slices* sucesivos de un *kernel* se lanzan en el mismo *stream*, un nuevo *slice* tiene que esperar a la finalización total del *slice* previo antes de empezar a ejecutarse. Sin embargo, durante la fase de finalización de los *slices*, se desperdician recursos de la GPU ya que algunos CTAs de los *slices* siguen ejecutándose mientras otros ya han terminado.
- *cCUDA* asigna dinámicamente los CTAs a los SMs utilizando un enfoque *greedy* que puede dar lugar a mapeados menos eficientes, mientras que *rtSMK* supervisa y controla constantemente la asignación de los CTAs.

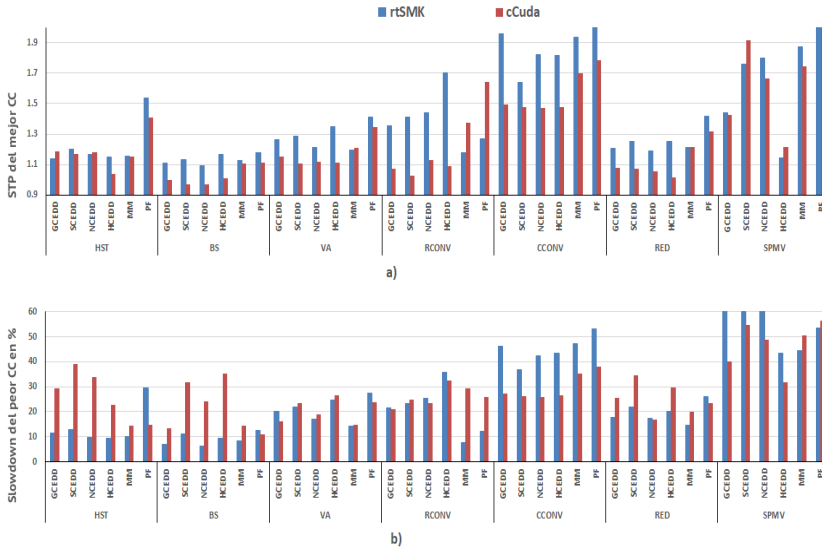


Figura 4.7: FlexSched versus *cCUDA*. a) muestra los *STP* máximos alcanzados por el mejor CC cuando se coejectutan los pares de *kernels* CB y MB utilizando nuestro esquema de planificación de CTAs (*rtSMK*) y el esquema de *slicing* de *cCUDA*. Se puede observar que nuestra técnica es capaz de obtener mejores resultados en la mayoría de casos. Así, el *STP* medio obtenido por *rtSMK* es de 1,40 mientras que el *STP* de *cCUDA* solo consigue 1,29. b) muestra el *slowdown* respecto al mejor *STP* cuando se utiliza la configuración CC que obtiene el menor *STP*. El *slowdown* medio de ambos métodos se sitúa en torno al 29%, lo que indica que la selección de un buen CC es primordial para obtener un buen rendimiento.

Como resultado, si calculamos el *STP* medio para todos los emparejamientos de *kernels*, *rtSMK* obtiene un valor de 1,40, mientras que *cCUDA* logra 1,29. Así, *rtSMK* es competitivo cuando se compara con las técnicas basadas en *slicing* de los *kernels*. Sin embargo, la ventaja más importante de nuestro método reside en el mecanismo de *preemption* incluido, que le permite encontrar la mejor configuración de coejecución durante una fase de *profiling* productivo, como ya se explicó en la sección 4.5.4. Por el contrario, *cCUDA* lanza los comandos de ejecución de los *slices* del *kernel* uno tras otro y se almacenan en una cola de comandos *hardware* asociada al *stream* correspondiente. En esta cola, los *slices* se ejecutan secuencialmente hasta que todos los comandos hayan terminado utilizando una planificación no preventiva.

La figura 4.7.b refleja la importancia de encontrar el mejor CC. En este gráfico de barras, el *slowdown* de la peor CC con respecto a la mejor se calcula mediante la ecuación 4.8, donde $STP_M(K1, K2)$ y $STP_m(K1, K2)$ son el *STP* máximo (M) y el mínimo (m) obtenidos para el mejor y el peor CC, respectivamente. Los valores del *slowdown* van del 6 % al 60 % en *rtSMK* y del 10 % al 40 % en *cCUDA*. El factor de *slowdown* medio para *rtSMK* y *cCUDA* se sitúa en torno al 29 %. Estos resultados muestran que se pueden obtener valores de rendimiento bajos si el CC no es adecuado.

$$SD(K1, K2) = \frac{STP_M(K1, K2) - STP_m(K1, K2)}{STP_M(K1, K2)} \cdot 100 \quad (4.8)$$

4.6.4. Rendimiento de la planificación *FlexSched*

FlexSched es un planificador que puede ejecutar simultáneamente un conjunto de aplicaciones aumentando el rendimiento por ejemplo en cuanto a tiempo de ejecución o para cumplir con unos requisitos de calidad de servicio. En esta sección se explica en detalle la heurística utilizada por *FlexSched* durante la fase de *profiling* para encontrar la configuración de coejecución que consigue el mejor rendimiento. A continuación, se valida la heurística en un escenario real en el que se ejecutan varias aplicaciones y se comparan los resultados con los obtenidos por el planificador *hardware* de la GPU (*HyperQ*).

- Fase de profiling.** Durante la primera etapa de la fase de *profiling* de *FlexSched*, se obtiene el CCS de los *kernels* que se ejecutan conjuntamente. Esta información puede recopilarse *off-line*, ya que solo requiere detalles sobre el uso de los recursos de los *kernels* que se establecen durante la compilación de los mismos. Más concretamente, esta información consiste en el número de hilos por CTA, el uso de memoria compartida por CTA y el número de registros utilizados por hilo. Por lo tanto, cualquier configuración válida en CCS se caracteriza por el número máximo de bloques persistentes que se pueden planificar de forma concurrente en un SM para ambos *kernels* a coejecutar, teniendo en cuenta la información anterior.

Una vez obtenido el CCS, *FlexSched* puede buscar una configuración adecuada, como se muestra en el ejemplo de la figura 4.8, donde se analiza un par de *kernels* (VA/RCONV). El CCS de este emparejamiento consta de siete configuraciones de coejecución, concretamente $CCS = (1,7), (2,6), (3,5), (4,4), (5,3), (6,2), (7,1)$, donde el primer y el segundo elemento de cada configuración indican el número de BSUs para VA y RCONV, res-

pectivamente. *FlexSched* pasa por este CCS, lanzando cada configuración para obtener los valores TER, y compara las configuraciones consecutivas utilizando el *speedup* ponderado (STP_S) calculado mediante la ecuación 4.4. Para mayor claridad, todos los valores de STP_S que atraviesan el CCS de izquierda a derecha (línea de puntos verde) y en sentido contrario (línea discontinua azul) se muestran en el gráfico de la izquierda de la figura 4.8. Por ejemplo, el punto más a la izquierda de la línea de puntos verde indica que la TER al pasar de la configuración (1,7) a la configuración (2,6) es de 1,38. Como se explica en la sección 4.5.4, un valor de TER mayor que uno significa que el valor de TER de la configuración (3,5) es mejor que el correspondiente de la configuración (2,6), que a su vez es mejor que el de TER de la configuración (1,7). Sin embargo, el TER de la configuración (4,4) es peor que el TER de la configuración (3,5), y las siguientes configuraciones también tienen valores de STP_S inferiores a uno, por lo que la mejor configuración para el par VA/RCONV es (3,5). Obsérvese que se puede obtener la misma conclusión al pasar por CCS en la dirección opuesta. Así, la mejor configuración se puede encontrar buscando la última tupla con un valor STP_S superior a uno, ya que obtiene el mayor valor TER de todas las tuplas CCS.

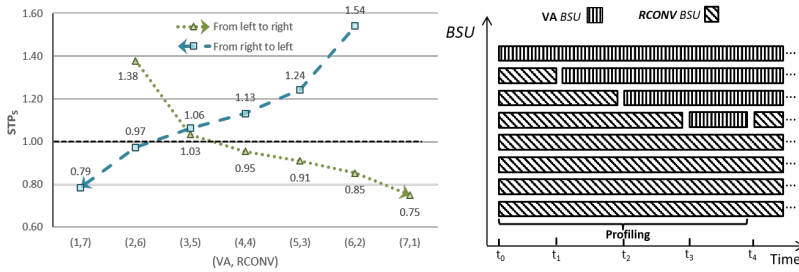


Figura 4.8: Inicio de *profiling* de *FlexSched*. El gráfico de la izquierda muestra el valor de STP_S para diferentes configuraciones de coejecución de VA y RCONV. La línea verde discontinua recorre el CCS de izquierda a derecha, mientras que la línea de puntos azul lo recorre en la dirección opuesta. El diagrama de la derecha muestra la fase de *profiling* hasta encontrar la mejor configuración. En primer lugar, se lanzan una BSU VA y siete BSUs RCONV en el tiempo t_0 . A continuación, se desaloja una BSU RCONV y se lanza una nueva BSU para VA en el tiempo t_1 . Este procedimiento continúa hasta que se encuentra la mejor configuración.

FlexSched no necesita obtener todos los valores de STP_S en ambas direc-

ciones. En su lugar, recorre CCS calculando sobre la marcha los valores que necesita hasta que se cruza la marca uno. En el ejemplo de la figura 4.8, *FlexSched*, siguiendo la línea verde discontinua, toma la primera configuración en CCS, (1,7), y lanza una BSU de VA y siete BSU de RCONV en el tiempo t_0 . Después de un tiempo de muestreo, se leen los valores de TER para cada *kernel* y se calcula su STP_S .

A continuación, *FlexSched* comprueba el rendimiento de la configuración (2,6) desalojando una BSU RCONV y lanzando una nueva BSU para VA en el tiempo t_1 . Este proceso se vuelve a realizar hasta que se alcanza la configuración (4,4) porque STP_S cae por debajo de 1. Después, *FlexSched* desaloja una BSU de VA y lanza una nueva BSU para RCONV para volver a la configuración (3,5) en el tiempo t_4 , dando por finalizada la fase de *profiling*. Esta configuración se mantiene hasta que uno de los dos *kernels* termina. En el peor de los casos para este método de búsqueda simple, el *profiler* debería evaluar $nconf - 1$ configuraciones, siendo $nconf$ el número de tuplas dentro de CCS. Sin embargo, computacionalmente la complejidad podría reducirse aún más utilizando un enfoque de “divide y vencerás” para el proceso de búsqueda.

La figura 4.9 muestra la situación cuando RCONV finaliza en el instante t_5 . Cuando *FlexSched* detecta la finalización de RCONV, selecciona otro *kernel*, MM en este ejemplo, y obtiene el CCS para esta pareja. Este CCS ya tiene una configuración con tres BSUs de VA, (5,3), por lo que *FlexSched* lanza cinco BSUs de MM en el instante t_5 . A continuación, en t_6 , se calculan los valores TER para la coejecución de estos *kernels* y lanza la siguiente configuración. Hay dos opciones, la configuración (4,4) o la configuración (6,2). Supongamos que se selecciona la configuración (6,2), con lo que se desaloja una BSU VA y se lanza una nueva BSU para MM. Los valores TER se calculan en el momento t_7 , y se calcula el STP_S entre las configuraciones (5,3) y (6,2). Tiene un valor superior a 1, concretamente 1,07, por lo que (6,2) es mejor y se puede comprobar la siguiente configuración, (7,1). En el tiempo t_8 , se calcula STP_S , obteniendo un valor inferior a 1 (0,96), por lo que se restablece la configuración anterior y se mantiene hasta que uno de los dos *kernels* finalice. En el tiempo t_6 *FlexSched* podría haber elegido la configuración (4,4), pero entonces en el tiempo t_7 habría obtenido $STP_S = 0,91$. Como es inferior a 1, *FlexSched* descartaría esta dirección y seleccionaría la configuración (6,2) (desaloja dos BSUs de VA y lanza dos BSUs de MM). A continuación, el proceso de búsqueda continúa de forma similar a lo indicado anteriormente.

Para algunos pares de *kernels*, los gráficos de STP_S pueden ser muy di-

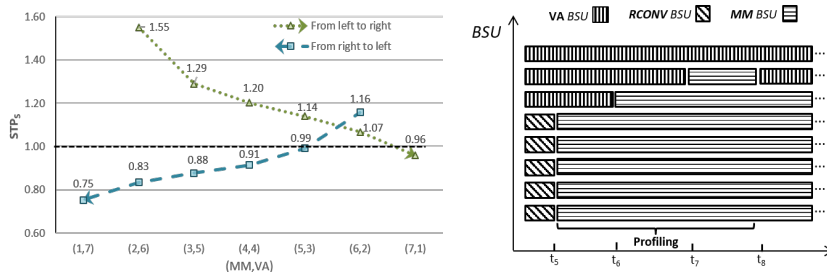


Figura 4.9: Fin del *profiling* de *FlexSched*. Al finalizar RCONV, nuestro planificador selecciona a MM y lanza 5 BSUs para llenar el espacio dejado por RCONV. A continuación, el *profiler* recorre el nuevo CCS para encontrar la mejor configuración de coejecución para VA y MM.

ferentes de los ejemplos anteriores, con valores siempre por encima o por debajo de uno. Por ejemplo, la figura 4.10 muestra los valores obtenidos para el par PF/SPMV. De izquierda a derecha están siempre por encima de 1 y, en consecuencia, de derecha a izquierda están por debajo de 1. Así, si *FlexSched* empieza a buscar por la izquierda, la configuración (1,14), lanzará todas las configuraciones hasta llegar a la última. Por otro lado, si *FlexSched* empieza a buscar desde la derecha, la configuración (7,2), entonces dejará de buscar tras comprobar que la siguiente configuración, (6,4), tiene un valor STP_s inferior a 1 (0,69).

Hemos realizado un experimento para demostrar que la configuración obtenida con esta heurística es óptima o casi óptima en todos los casos. En primer lugar, hemos obtenido las configuraciones óptimas para todos los pares de *kernels* utilizando el método de fuerza bruta introducido en la sección 4.6.3. A continuación, se han comparado los resultados de rendimiento de estas configuraciones óptimas con los resultados de rendimiento de las configuraciones seleccionadas por *FlexSched*. La diferencia media relativa es inferior al 1%, es decir, *FlexSched* puede encontrar la mayoría de las veces la mejor configuración, o cerca de la mejor. Las diferencias más elevadas se obtienen para las configuraciones de ejecución concurrente con bajo rendimiento. Por ejemplo, el error máximo alcanza el 6% cuando se coejecutan RCONV y RED. Ambos *kernels* están limitados por la memoria y la mejor configuración de coejecución obtiene un valor de STP_s de 1,0 respecto a la ejecución secuencial, mientras que *FlexSched* encuentra una configuración que alcanza el 0,94.

- **Coste del *profiling*.** Durante la fase de *profiling*, *FlexSched* cambia di-

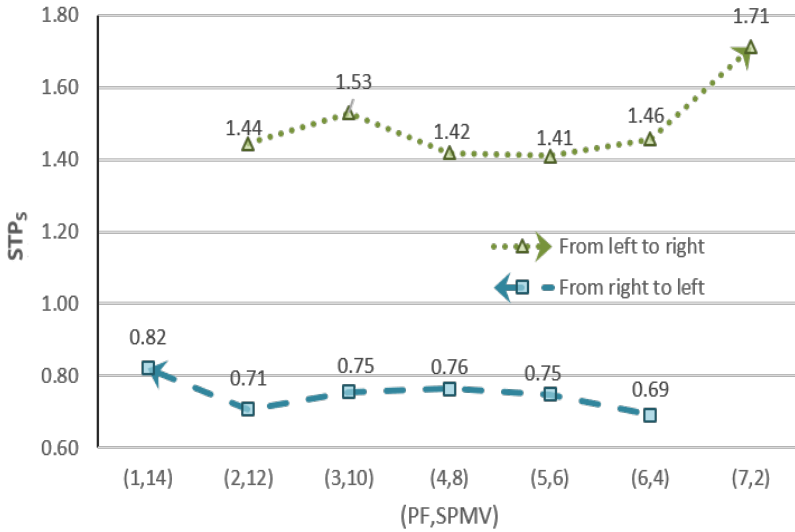


Figura 4.10: *Profiling* de *FlexSched* para PF/SPMV. Valores de STP_S tomados para el CCS de PF/SPMV. Todos los valores de izquierda a derecha son mayores que uno.

namicamente el mapeo de las BSUs para encontrar una configuraci3n de coejecuci3n con el mejor rendimiento. Sin embargo, la flexibilidad que ofrece el esquema de *profiling on-line* tiene un coste temporal, ya que debe aplicarse cada vez que hay que evaluar un CC. En este apartado vamos a evaluar el coste de esta fase de *profiling*.

Sea $T_{prof}(K_1, K_2)$ el tiempo que tarda la fase de *profiling* en encontrar, para un par de *kernels* especıficos (K_1, K_2) , la mejor configuraci3n. Este tiempo depende del numero de configuraciones que deban evaluarse antes de llegar a la mejor. Ademas, el tiempo empleado para cada configuraci3n requiere desalojar varias BSUs y lanzar otras nuevas. Como se muestra en la tabla 4.3, el tiempo de desalojo es diferente para cada *kernel*. En consecuencia, una evaluaci3n precisa del coste de la fase de *profiling* requiere evaluar este tiempo para todos los pares de *kernels* posibles.

Dado que esta fase de *profiling* se emplea para encontrar una buena configuraci3n de coejecuci3n que mejore a la ejecuci3n secuencial de los *kernels*, el coste puede calcularse comparando $T_{prof}(K_1, K_2)$ con la ejecuci3n secuencial de los *kernels*, calculando el mismo numero de subtareas por *kernel* que

las ejecutadas durante la fase de *profiling*. Es decir, el coste de la fase de *profiling* $O_{prof}(K_1, K_2)$ viene dado por la ecuación 4.9, donde $T_{alone}(K_i|task_i)$ es el tiempo que tardará el *kernel* K_i en ejecutar las subtareas ($task_i$) de forma secuencial, y $task_1$ y $task_2$ son el número de subtareas ejecutadas durante la fase de *profiling* para los *kernels* K_1 y K_2 , respectivamente.

$$O_{prof}(K_1, K_2) = \frac{T_{prof}(K_1, K_2)}{T_{alone}(K_1|task_1) + T_{alone}(K_2|task_2)} \quad (4.9)$$

La figura 4.11a muestra, mediante un *boxplot*, la distribución estadística de O_{prof} para todos los pares posibles de *kernels* (78 valores han sido recolectados). Se puede observar que, aunque algunos pares de *kernels* obtienen valores superiores a uno, la mayoría de ellos están por debajo de este valor. Por lo tanto, en la mayoría de los casos, el tiempo empleado durante la fase de *profiling* es inferior al tiempo de la ejecución secuencial de un número similar de subtareas. De hecho, la media y la mediana para O_{prof} son 0,913 y 0,926, respectivamente. Una conclusión importante que se puede extraer de este experimento es que, durante la fase de *profiling*, a pesar de las operaciones de desalojo y lanzamiento de BSUs, el coste de estas operaciones queda oculto gracias a la ejecución concurrente de otras BSUs, lo que da lugar a un tiempo de ejecución total que suele ser más corto que el cómputo equivalente realizado por la ejecución secuencial de los *kernels*. Por tanto, no es necesario hacer un estudio del rendimiento de la ejecución secuencial de cada *kernel* para obtener una buena configuración de coejecución.

- **Ejecución concurrente de aplicaciones.** En este experimento se compara *FlexSched* con el mecanismo de planificación *hardware* nativo proporcionado por NVIDIA (HyperQ). Como nos interesa analizar el comportamiento de nuestro planificador en un escenario real, los experimentos ejecutan las 9 aplicaciones (13 *kernels*) indicados en la tabla 4.2. Otros trabajos, como [100, 104], solo ejecutan dos *kernels* así que vamos a evaluar el rendimiento alcanzado al coejecutar parejas de *kernels*. También asumimos que todas las aplicaciones están listas para ejecutarse (todos los datos de entrada necesarios se han transferido del *host* al dispositivo) antes de que comience la planificación.

En este experimento, *FlexSched* comienza seleccionando uno de los 13 *kernels*, luego examina su categoría, como se indica en la tabla 4.2, y selecciona al azar una pareja entre los *kernels* de las demás categorías. A continuación, durante la fase de *profiling*, busca una buena configuración de coejecución como se explica en el punto anterior. Cuando cualquiera de los dos *kernels* termina, un nuevo *kernel* que pertenece a otra categoría se selecciona, si

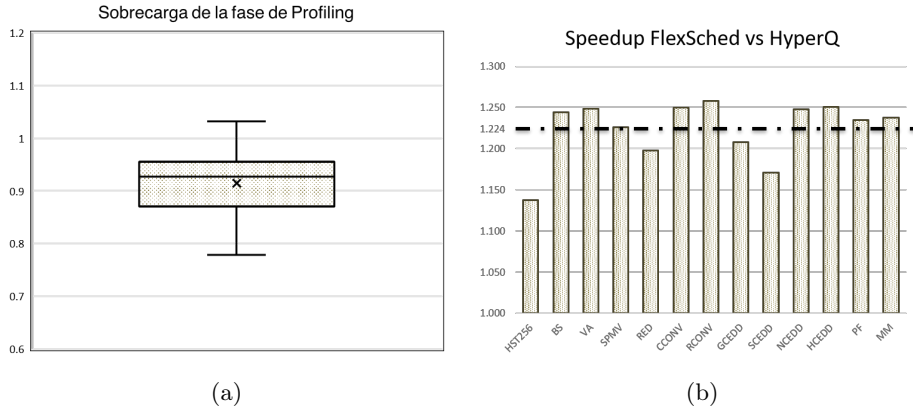


Figura 4.11: (a) *Boxplot* de *profiling*. Muestra el coste de la fase de *profiling* para todos los pares de *kernels* posibles. El coste se calcula respecto a la ejecución secuencial del mismo número de subtareas. (b) *Speedup* de *FlexSched* versus *HyperQ*. Cada barra se corresponde con el lanzamiento en primer lugar de uno de los 13 *kernels*.

está disponible. En caso contrario, se elige un *kernel* de la misma categoría. Cualquier planificación de un nuevo *kernel* implica una fase de *profiling* para buscar una buena configuración. Este proceso continúa hasta que se termina con todas las aplicaciones. La selección de los *kernels* que se ejecutan conjuntamente es aleatoria, por lo que ejecutamos el experimento 1000 veces para realizar un análisis estadístico del comportamiento del planificador. Además, repetimos este experimento utilizando cada uno de los 13 *kernels* como primer *kernel* para tener en cuenta las diferencias.

El mismo experimento se ha ejecutado utilizando *HyperQ*, una característica de las GPU modernas desde la arquitectura Kepler, en la que varias colas gestionadas por el *hardware* pueden planificar los comandos del *kernel* desde diferentes *streams*. Así, cada aplicación se lanza al mismo tiempo en un *stream* diferente. Para hacer una comparación justa, cada experimento se ejecuta de nuevo 1000 veces, y los *streams* se lanzan en el mismo orden en el que se ejecutaron las aplicaciones en el experimento correspondiente con *FlexSched*. Para cada ejecución, hemos calculado el *speedup* del tiempo total de ejecución utilizando *FlexSched* con respecto a *HyperQ*. En la figura 4.11b se muestra la media de estos valores cuando se utiliza cada *kernel* como primer *kernel*, y se dibuja una línea horizontal para indicar la media total. La heurística de *FlexSched* obtiene diferentes valores de

speedups en función del primer *kernel*, ya que se eligen diferentes pares de coejecución, pero explota la capacidad de ejecución concurrente de la GPU mucho mejor que HyperQ, logrando un *speedup* medio de 1,224. HyperQ obtiene siempre el peor rendimiento, ya que solo puede solapar la ejecución de un *kernel* nuevo cuando el anterior está terminando.

4.6.5. Planificación orientada a la latencia con *FlexSched*

En un centro de procesamiento de datos, las aplicaciones pueden tener diferentes requisitos; por ejemplo, puede haber, al mismo tiempo, un lote de aplicaciones orientadas al rendimiento junto con aplicaciones sensibles a la latencia (LS) que deben ejecutarse dentro de un tiempo de respuesta máximo. Una estrategia *naïve* para planificar una de estas aplicaciones LS es desalojar todos los *kernels* que se estén ejecutando en ese momento y luego lanzar la aplicación LS. Sin embargo, este enfoque desperdiciará recursos de la GPU que podrían utilizarse para coejecutar otros kernels mientras se cumplen las restricciones temporales de la aplicación LS.

FlexSched puede ampliarse para soportar la coexistencia de aplicaciones LS con un lote de aplicaciones orientadas al rendimiento. A diferencia de [104], que solo puede imponer restricciones en cuanto al número de CTAs asignados a las aplicaciones LS, nosotros podemos emplear una restricción temporal más útil basada en el valor mínimo de TER permitido. Así, cuando una aplicación LS se coejecuta con otros *kernels*, *FlexSched* busca, durante la fase de *profiling*, configuraciones que garanticen este valor de TER. Más concretamente, *FlexSched* comienza seleccionando la configuración en CCS que asigna más BSUs a la aplicación LS. A continuación, calcula su valor de TER y, si es mayor que el mínimo permitido, se selecciona la siguiente configuración en CCS. Cuando no se cumple la restricción de la tasa de subtareas, se restablece la configuración anterior y finaliza la fase de *profiling*.

El experimento para probar el comportamiento de *FlexSched* con aplicaciones LS consiste en ejecutar concurrentemente un *kernel* con restricciones temporales con un lote de *kernels*. Para obtener una variada distribución de experimentos, definimos varios factores de *slowdown* de la TER concurrente con respecto a la TER secuencial de esa aplicación cuando se ejecuta sola. Estos factores varían de 0,5 a 0,1, en pasos de 0,1. Por ejemplo, un *slowdown* de 0,5 significa que el valor mínimo permitido de TER de la aplicación LS debe ser al menos la mitad del valor de TER cuando se ejecuta sola. Además, hemos seleccionado el *kernel* CEDD como aplicación LS, y lo hemos ejecutado de forma concurrente con

otros *kernels*. En caso de que los otros *kernels* terminen antes que la aplicación LS, se lanzan de nuevo. La tabla 4.4 muestra los resultados cuando CEDD se ejecuta de forma concurrente con varios *kernels* limitados por la memoria o por la unidades de cómputo. Para cada *slowdown*, registramos el porcentaje de CTAs que pertenecen al otro *kernel* que puede ser coejecutado con el *kernel* CEDD mientras se cumplen los requisitos de rendimiento de LS. Así, cuando MM se ejecuta simultáneamente con CEDD imponiendo un *slowdown* máximo de 0,5, el 31 % de los CTAs que se ejecutan en la GPU pertenecen a MM. También se muestran los valores medios del porcentaje de CTAs para cada *slowdown*. Como era de esperar, cuando el *slowdown* disminuye, el número de los demás CTAs del *kernel* que pueden ser coejecutados con CEDD decrece, ya que hay que asignar más recursos a la aplicación LS. Sin embargo, incluso para valores de *slowdown* pequeños, nuestro planificador es capaz de asignar recursos al otro *kernel*. En conclusión, *FlexSched* puede utilizarse para planificar aplicaciones que deben cumplir estrictos requisitos de calidad del servicio (QoS) y, además, lanzar otros *kernels* para lograr un alto uso de los recursos.

	Slowdown factor				
	0.5	0.4	0.3	0.2	0.1
MM	31	23	13	11	1
VA	28	21	12	7	1
BS	18	12	7	1	1
RED	37	28	23	12	1
PF	40	33	22	12	1
HST256	33	22	18	12	1
Average	31	23	16	9	1

Tabla 4.4: Porcentaje de CTAs por SM usados por los *kernels* no sensitivos a la latencia respecto al total de CTAs disponibles durante la ejecución concurrente con el *kernel* CEDD.

4.7. Resumen

En este capítulo hemos presentado un modelo *software* de ejecución concurrente de *kernels* en una GPU. Este modelo, denominado *FlexSched*, implementa políticas de planificación destinadas a maximizar el rendimiento en la ejecución de los *kernels* o a satisfacer requisitos de calidad de servicio (QoS) de la misma, como por ejemplo el tiempo máximo de respuesta.

Nuestro planificador requiere de unas modificaciones mínimas en el código del

kernel que nos habilitan para hacer una distribución específica de los recursos de la GPU. El sistema incluye un módulo encargado de planificar los *kernels* para su coejecución, Dicho módulo utiliza los datos recolectados durante una fase de *profiling* productivo para realizar una buena distribución de los recursos. Una característica importante de *FlexSched*, que lo diferencia de planificadores *software* propuestos en otros trabajos, es el uso de este *profiler on-line* productivo. Durante la fase de *profiling*, nuestro planificador emplea una heurística que compara diferentes configuraciones de coejecución para encontrar un esquema de planificación de CTAs adecuado que cumpla los requisitos de planificación: rendimiento o QoS. Gracias a este esquema de planificación flexible, *FlexSched* puede aplicarse a situaciones reales en las que *kernels* desconocidos deben ejecutarse inmediatamente en una GPU, a diferencia de trabajos anteriores que necesitan realizar *profiling off-line* muy costosos.

En la validación de nuestro modelo *software* se han realizado numerosos experimentos utilizando 9 aplicaciones con 13 *kernels* en total. En primer lugar, para comparar nuestra propuesta con otros enfoques para la coejecución de *kernels*, hemos confrontado rtSMK, el planificador de CTAs que hemos desarrollado para *FlexSched*, con un método de *slicing*, cCUDA [85], obteniendo valores de *STP* entre 1,1 y 2,1 y una mejora media en torno al 10% con respecto a cCUDA. También hemos realizado un experimento que muestra que el aumento en el *STP* logrado por la coejecución de los *kernels* depende en gran medida de seleccionar una planificación de coejecución adecuada. A continuación, hemos demostrado que la heurística empleada por el planificador *FlexSched* es capaz de encontrar la planificación de coejecución que alcanza el mayor rendimiento en la mayoría de los casos. Además, la pérdida media de rendimiento con respecto a la mejor planificación posible es inferior al 1%. *FlexSched* también se ha comparado con el planificador *hardware* que utiliza HyperQ. El experimento, a diferencia de otros trabajos en los que solo se realiza una planificación de coejecución específica, considera simultáneamente todas las aplicaciones. Así, durante el experimento, *FlexSched* ha buscado configuraciones de coejecución adecuadas a medida que se lanzaban nuevos *kernels* además de ejecutarlos. No obstante, *FlexSched* ha mejorado el rendimiento de los *kernels* en un factor de 1,22 con respecto a HyperQ. Por último, se han realizado experimentos con aplicaciones sensibles a la latencia que requieren un tiempo de respuesta estricto. En este caso, *FlexSched* ha calculado cuántos recursos de la GPU podrían asignarse a las aplicaciones por lotes sin dejar de cumplir los requisitos de la aplicación sensible a la latencia. Una vez más, hemos demostrado que el proceso de *profiling* es crucial para aprovechar el uso de los recursos de la GPU mientras se cumplen las restricciones de la calidad del servicio (QoS).

5 Planificación eficiente de kernels usando técnicas hardware

En este capítulo se aborda el problema de la ejecución concurrente de *kernels* (CKE) mediante mecanismos *hardware*. La planificación eficiente de un conjunto de *kernels* para su coejecución requiere de mecanismos *hardware* que permitan distribuir los bloques de los *kernels* sobre los *Streaming Multiprocessor* (SM) de la GPU. Además, esta planificación se puede orientar a la consecución de objetivos de *fairness* y de calidad de servicio (*QoS*) de forma eficiente.

En la sección 5.1 se muestra el estado del arte referente a la ejecución concurrente de *kernels* en la GPU mediante el uso de mecanismos *hardware*. Un ejemplo motivador que muestra las ventajas de la coejecución de *kernels* en una GPU se ilustra en la sección 5.2. La sección 5.3 describe el modelo *hardware* planteado, *HPSM*, para encontrar la mejor planificación de dos *kernels* coejecutados. La implementación de nuestro modelo se describe en la sección 5.4. En la sección 5.5 se muestran los experimentos realizados para validar nuestro modelo. Finalmente, en la sección 5.6 se hace un resumen de las conclusiones obtenidas para nuestro modelo *hardware*.

5.1. Estado del Arte

La ejecución concurrente de *kernels* mediante mecanismos *hardware* ha sido ampliamente estudiada. Uno de los primeros trabajos en proponer modificaciones

hardware para soportar CKE compara la distribución de varios *kernels* sobre los SMs usando una técnica de *spatial multitasking* (SMT) con otra técnica denominada *cooperative multitasking* donde se comparten los recursos dentro de un SM entre los *kernels* a coejecutar (equivalente a la técnica de *simultaneous multikernel* o SMK). En ese trabajo [1] se muestran las ventajas de SMT respecto a SMK. Xu Q. et al. [103] proponen una política de planificación de CTAs *intra-SM* o SMK. Este trabajo se centra en la reducción de la fragmentación de los recursos cuando se asignan CTAs de distintos *kernels* en un SM y también proponen un método de partición de recursos que maximiza el rendimiento. Sin embargo, requiere de una fase de *profiling off-line* para obtener el valor de IPC de cada *kernel*. En esta fase de *profiling off-line* varían el número de CTAs por SM para cada *kernel*, por lo que encontrar una distribución adecuada de CTAs puede llevar un periodo de tiempo prolongado. Para reducir este tiempo, proponen realizar estas mediciones durante la coejecución de varios *kernels*.

Zhao et al [105] realizan un estudio en profundidad de los métodos *hardware* que existen para la ejecución concurrente de *kernels*, incluyendo los mecanismos de *preemption* propuestos hasta el momento. Los mecanismos de *preemption* tienen como objetivo desalojar *kernels* con baja prioridad para poder asignar los recursos a *kernels* con prioridad alta, reduciendo el tiempo de espera para estos últimos. Los sistemas de *preemption* propuestos incluyen el cambio de contexto [51], el drenado de SMs [92] y el purgado de SMs [78]:

- En el cambio de contexto se guarda el contexto del *kernel* en ejecución y se liberan los recursos de la GPU para asignárselos al *kernel* de mayor prioridad. En el cambio de contexto es necesario guardar el estado de los registros, la pila SIMT, así como el estado de los *warps* o CTAs desalojados con el fin de restaurarlos adecuadamente al retomar su ejecución. En el caso de los CTAs también se debe guardar el estado de la memoria compartida y de las barreras de sincronización. En comparación con el cambio de contexto en una CPU, el cambio de contexto en una GPU es muy elevado, lo que supone una gran sobrecarga. La sobrecarga y la latencia producidas por el mecanismo de *preemption* al no poder ejecutar nada durante el cambio de contexto influyen negativamente en el rendimiento del sistema.
- En el drenado de SMs se lanzan CTAs del *kernel* prioritario a los SMs después de que todos los CTAs del *kernel* en ejecución hayan terminado, lo que puede reducir la sobrecarga producida por el cambio de contexto. En este caso, la latencia solo depende del tiempo de ejecución restante de los CTAs del *kernel* en ejecución. La desventaja es que el tiempo de la *preemption* puede ser muy elevado.

- En el purgado de SMs se desalojan los CTAs del *kernel* en ejecución sin guardar el contexto y se lanza el *kernel* prioritario. Al terminar la ejecución del *kernel* prioritario, los CTAs desalojados pueden volver a ejecutarse desde el principio. El principal problema es que este método solo funciona con *kernels* idempotentes, es decir, *kernels* que no tienen operaciones atómicas y que no sobrescriben posiciones de memoria global que se lean. Además, aunque la latencia de este mecanismo es casi nula, el trabajo útil se desperdicia cuando un bloque se desaloja.

Wang Z. et al. [98] proponen una estrategia de planificación de CTAs *intra-SM* o SMK, al que le añaden un mecanismo de *preemption*. De este modo, pueden lanzar un *kernel* prioritario tras desalojar los CTAs pertenecientes al *kernel* en ejecución. Gracias a este mecanismo de *preemption*, se pueden emplear estrategias para mejorar el rendimiento general y realizar un reparto justo entre los *kernels* durante la coejecución. Park J.J.K. et al. [78] buscan minimizar el coste en el cambio de contexto al desalojar un *kernel* de la GPU.

Park J.J.K. et al. en [79] proponen un mecanismo productivo para encontrar la mejor planificación de CTAs en la coejecución de dos *kernels*. Para explorar de forma rápida todas las combinaciones posibles, proponen una heurística en la que se asignan diferentes distribuciones de CTAs a distintos grupos de SMs. Analizando los valores de IPC de los SMs, encuentra gradualmente la mejor distribución de CTAs para la coejecución de los *kernels*. Para reducir la sobrecarga producida por la búsqueda de la mejor distribución de los recursos entre los *kernels* a coejecutar, Zhao W. et al. [106] usan un predictor entrenado para conocer el *slowdown* de los *kernels* coejecutados. Este predictor recoge las estadísticas de los eventos *hardware* de los dos *kernels* a coejecutar y estima su *slowdown*.

En [23, 97] se centran en el subsistema de memoria con el fin de aumentar el rendimiento de los *kernels* a coejecutar. Los primeros tratan de evitar la inanición de los *kernels* que hacen un uso intensivo de los recursos de cómputo cuando se ejecutan junto con *kernels* que hacen un uso intensivo de la memoria. Observaron que, en esta configuración, la latencia de los accesos a la memoria global de los *kernels* que hacen un uso intensivo de los recursos de cómputo crece. Esto se debe a que sus peticiones de memoria se ponen en cola detrás del gran número de peticiones emitidas por los *kernels* que hacen un uso intensivo de la memoria. Para permitir el acceso a memoria de los *kernels* que hacen un uso intensivo de los recursos de cómputo desarrollan métodos para retrasar los accesos a la memoria de los *kernels* que hacen un uso intensivo de la memoria. Esto incrementa el rendimiento del sistema y el *fairness* utilizando una métrica que tiene en cuenta tanto el ancho de banda de la memoria DRAM como las tasas de fallo de la caché.

Zhao X. et al. [107] utilizan una planificación *intra-SM* o SMK para la coejecución de *kernels*. El trabajo muestra un método para clasificar los *kernels* (limitados por la memoria o limitados por el cálculo) mientras los *kernels* se ejecutan concurrentemente utilizando contadores de la GPU tanto para el número de accesos a la memoria, como para las consultas al *row buffer* de la DRAM. Además, se puede realizar una estimación del rendimiento de un solo *kernel* que permite evaluar el rendimiento de la coejecución con respecto a la ejecución secuencial.

5.2. Motivación

La arquitectura de la GPU puede hacer que algunos *kernels* no escalen adecuadamente por múltiples motivos. Así, el ancho de banda de la memoria global puede verse saturado con *kernels* que hacen un uso intensivo de la memoria antes de que se alcance el número máximo de CTAs posibles para ese *kernel* dentro de un SM. Las *pipelines* son otros recursos que pueden verse saturados por los *kernels* que hacen un uso intensivo de los recursos de cómputo. Otros *kernels* pueden saturar la caché L1 [103]. Por tanto, una estrategia útil para aumentar el uso de la GPU puede venir de la mano de una reducción del número de CTAs asignados a los *kernels* que saturan los recursos de la GPU, permitiendo, al mismo tiempo, el lanzamiento de CTAs de otro *kernel*. De esta forma se puede aumentar el rendimiento global del sistema.

En el siguiente ejemplo se ejecutan dos *kernels* independientes en el simulador GPGPU-Sim en su versión 4.0 [41], al cual le hemos añadido soporte para la ejecución concurrente de *kernels* (CKE). El archivo de configuración utilizado para configurar la GPU ha sido el de la Titan V de la arquitectura Volta, el cual modela una GPU con 40 clusters de 2 cores cada uno, sumando un total de 80 SMs, conectados a una memoria HBM de 3 *stacks* y 24 canales. En cuanto a los *kernels* utilizados, GCEDD, es un *kernel* intensivo en cómputo, y RED es un *kernel* intensivo en memoria. Para coejecutarlos se pueden aplicar distintas políticas de planificación de los recursos de cómputo como SMT y SMK. En las figuras 5.1a y 5.1b se muestra la ejecución independiente de estos dos *kernels* aplicando distintas distribuciones de CTAs en cada política, línea azul para una política SMK y línea dorada para una política SMT. En ambas políticas se aumentan gradualmente el porcentaje de recursos de cómputo asignados a cada *kernel*. Mas concretamente, para SMT se aumentan el número de SMs, mientras que para SMK se aumentan el número de CTAs dentro de cada SM. En la figura 5.1a se muestra el IPC alcanzado por GCEDD a medida que se utilizan más recursos en el dispositivo. Se puede observar que los valores de IPC de la ejecución SMT de

GCEDD son lineales con el número de SMs [107]. Sin embargo, cuando se emplea una planificación SMK, los valores de IPC alcanzados para el mismo porcentaje de utilización de recursos son mayores. Esto indica que la planificación SMK hace un mejor uso de los recursos del SM en estas situaciones. En una planificación SMT se lanzan todos los CTAs posibles del *kernel* en los SMs asignados y el IPC obtenido por cada SM está limitado por el uso intensivo de los recursos compartidos pudiendo producir cuellos de botella dentro del SM. Sin embargo, en una planificación SMK se distribuyen los CTAs de manera uniforme entre todos los SMs, manteniendo bajo el número de CTAs en ejecución dentro de cada SM y, en consecuencia, se reduce la presión sobre los recursos.

Del mismo modo, la figura 5.1b muestra el IPC obtenido por un *kernel* intensivo en memoria, en este caso RED. Utilizando ambos esquemas de planificación, el *kernel* consigue un IPC similar en la zona de saturación, lo que indica que el *kernel* ha alcanzado su máximo ancho de banda en memoria. Sin embargo, en la zona de no saturación, la planificación SMK obtiene valores de IPC más elevados, ya que la ejecución de las instrucciones de acceso a la memoria se distribuye entre todos los SMs disponibles en lugar de restringirse a un subconjunto de estos SMs, lo que satura los *buffers* de *load/store* del SM.

Al coejecutar estos dos *kernels* se producen cambios en el comportamiento de ambos *kernels*. Este cambio de comportamiento en los *kernels* se debe a la interferencia producida al compartir los recursos de la GPU. Una métrica de rendimiento para medir la interferencia producida durante la coejecución de los varios *kernels* es *STP* (*System Throughput*). La expresión de *STP* para un conjunto de K *kernels* coejecutándose viene dada por la ecuación 2.2, la cual se repite a continuación (ecuación 5.1) para facilitar su referencia. En esta ecuación IPC_k^{shared} y IPC_k^{alone} indican el número de instrucciones por ciclo obtenido por el *kernel* k cuando se coejecuta con otros *kernels* y el número de instrucciones por ciclo cuando se ejecuta solo (todos los recursos de la GPU son asignados al mismo *kernel*), respectivamente. Cada término de la suma se denomina *NP* (Normalized Progress), y mide la pérdida de rendimiento (*slowdown*) del *kernel* k cuando se coejecuta con respecto a cuando se ejecuta solo.

$$STP = \sum_{k=1}^K \frac{IPC_k^{shared}}{IPC_k^{alone}} \quad (5.1)$$

La figura 5.1c ilustra el *STP* alcanzado por RED y GCEDD (cuanto más alto, mejor). En el eje de abscisas se indica el valor normalizado de CTAs lanzados por RED. GCEDD está utilizando los recursos restantes de la GPU para lanzar la mayor cantidad de CTAs posibles. El *STP* va a depender no sólo de los recursos

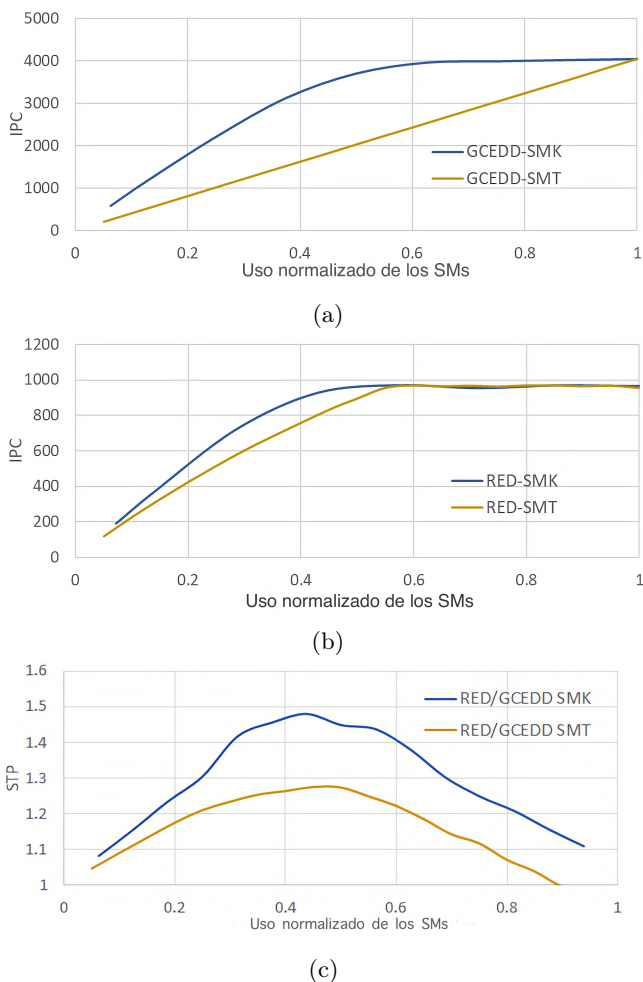


Figura 5.1: IPC alcanzado por la ejecución individual de los *kernels* (a) GCEDD y (b) RED utilizando un número variable de CTAs mediante SMT y SMK. (c) Rendimiento del sistema (*STP*) obtenido durante la coejecución de los *kernels* RED y GCEDD para diferentes políticas de planificación de los recursos SMT y SMK.

de SM asignados, sino también de las interferencias entre los *kernels* durante su coejecución causadas por el uso de otros recursos compartidos (unidades funcionales, *buffers* de *load/store*, *row buffers* de memoria global, etc.). Así, en el caso

de SMT, en el que todos los recursos del SM se asignan a un *kernel*, la interferencia aparece a nivel de la memoria global, ya que los *kernels* coejecutados van a competir por los recursos de memoria global. Con SMK, además de los conflictos producidos en la memoria global, también aparecen interferencias dentro del SM, ya que los CTAs de los distintos *kernels* se ejecutan dentro de un mismo SM. Comparando los resultados obtenidos para los esquemas de planificación SMT y SMK, podemos observar que SMK obtiene rendimientos más elevados que para SMT. Siguiendo el razonamiento que expusimos para explicar los resultados de las figuras 5.1a y 5.1b, podemos argumentar que SMK obtiene mejores resultados porque los CTAs del mismo *kernel* situados en el mismo SM no compiten entre ellos tan intensamente como en el esquema SMT.

También se puede observar que *STP* toma valores muy diferentes en función del número de CTAs que han sido ejecutados por cada *kernel*. Así, los valores de *STP* oscila entre 1,0 a 1,36 para SMT (línea dorada), y de 1,0 a 1,47 para SMK (línea azul). Sin embargo, la planificación que obtiene un mayor *STP* no se conoce de antemano, ya que depende de la interferencia entre los *kernels* al compartir recursos como la memoria global, las cachés L1 y L2, los *buffers* de *load/store*, las unidades funcionales, etc.

Esto demuestra las ventajas que aporta la coejecución de varios *kernels* al compartir los recursos de una GPU, y la necesidad de desarrollar un modelo de predicción que permita conocer el *slowdown* para que el planificador encuentre la mejor asignación de recursos de cómputo que cumpla con los objetivos específicos de la política de planificación aplicada.

La implementación de políticas de planificación para obtener un buen *slowdown* y mejorar el *fairness* en la coejecución de *kernels* requiere conocer el IPC en tiempo de ejecución de la coejecución de dos *kernels* (IPC^{shared}) y la ejecución en solitario de cada *kernel* (IPC^{alone}). En una implementación *hardware*, el valor de IPC^{shared} puede obtenerse fácilmente leyendo algunos de los contadores *hardware* internos del dispositivo. Sin embargo, el valor de IPC^{alone} no puede calcularse directamente y debe predecirse utilizando algún modelo.

La mayoría de modelos de planificación para la coejecución de *kernels* que buscan mejorar el *slowdown* recurren a aproximaciones que carecen de la precisión necesaria para predecir el progreso normalizado NP de una distribución de recursos de cómputo específica [79, 98, 103, 106]. En [107] se realiza una planificación SMT para coejecutar los *kernels* y el modelo *slowdown* se genera utilizando únicamente los valores extraídos durante la coejecución de los *kernels*. Este modelo, denominado Hybrid Slodown model (HSM), predice el ancho de banda de la memoria de cada *kernel* cuando utiliza la GPU en solitario. Esta predicción se

basa en la suposición de que el ratio de acierto en el *row buffer* para un *kernel* permanece constante para cualquier asignación de CTAs. En el caso de un *kernel* clasificado como de cómputo, se emplea un modelo lineal simple basado en los CTAs asignados al *kernel*.

5.3. Hybrid Slowdown Model (HSM)

Zhao et al. [107] propusieron un modelo de *slowdown* híbrido (HSM) que predice el NP para los *kernels* tanto limitados por la memoria, como los *kernels* limitados por las unidades de cómputo que se ejecuta al mismo tiempo bajo un esquema de planificación SMT. Los autores se dieron cuenta de que la utilización efectiva del ancho de banda de un *kernel* limitado por la memoria puede predecirse utilizando el ratio de aciertos en el *row buffer* (RBH) obtenido de los accesos a la memoria global. El *row buffer*, mostrado en la Figura 5.2, es el componente de la memoria DRAM donde se almacena la última fila de datos leída del banco de memoria. Los accesos posteriores a direcciones de la misma fila requieren un tiempo de acceso mucho menor, puesto que la fila ya está en el *row buffer* y no es necesario traerla de nuevo [30]. Esto es conocido como un acierto en el *row buffer*.

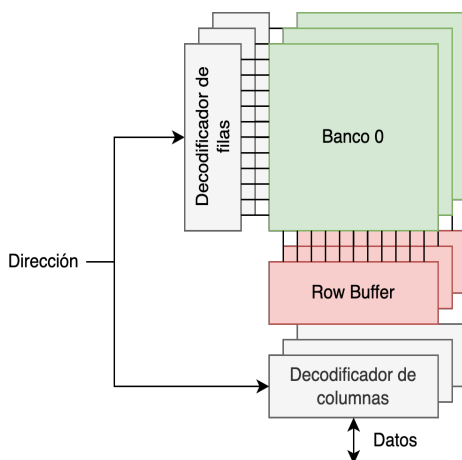


Figura 5.2: Memoria DRAM. Cada banco de memoria posee su decodificador de fila, su *row buffer* y su descodificador de columna. Con la fila seleccionada y la columna, se accede al dato buscado.

El valor NP de los *kernels* limitados por la memoria puede obtenerse mediante una regresión lineal y el valor RBH obtenido mientras se coejejuta con otro *kernel*. Utilizan este modelo para implementar una política de planificación consciente del *fairness* (HSM-Fair).

Para comprobar la validez de esta hipótesis en una planificación SMK, se ha ejecutado en solitario los *kernels* de la tabla 5.3 y se ha medido el ancho de banda y el RBH . En la figura 5.3 se muestra la línea de regresión calculada. En esta figura se muestra la posición de cada *kernels* con sus medidas del ancho de banda y RBH , mostrando que el modelo de regresión lineal HSM se ajusta bien a estos datos.

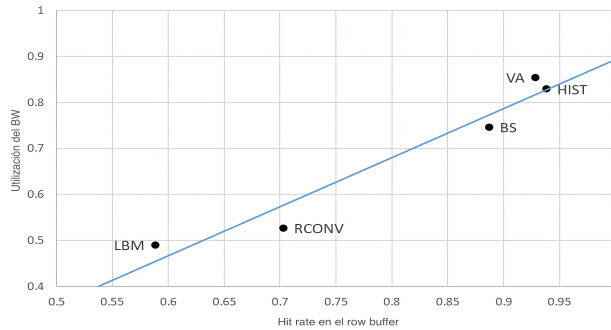


Figura 5.3: Ancho de banda y ratio de acierto en el *row buffer*. La predicción del ancho de banda (BW) requiere calcular, de antemano, un modelo lineal con la relación lineal entre el ratio de acierto del *row buffer* y la utilización del ancho de banda (BW).

El NP de un *kernel* de cómputo puede predecirse mediante una regla proporcional dada por la ecuación 5.2, donde IPC_{CB}^{shared} es el IPC obtenido por el *kernel* de cómputo cuando se coejejuta con otro *kernel*, R_{CB}^{alone} y R_{CB}^{shared} son los recursos de cómputo asignados a el *kernel* durante la ejecución solo y durante la coejejecución, respectivamente. Como HSM aplica una planificación SMT, R_{CB}^{alone} corresponde al número total de SMs existentes en la GPU y R_{CB}^{shared} es el número de SMs asignados al *kernel* de cómputo. Por lo tanto, NP para el *kernel* de cómputo viene dado por la ecuación 5.3.

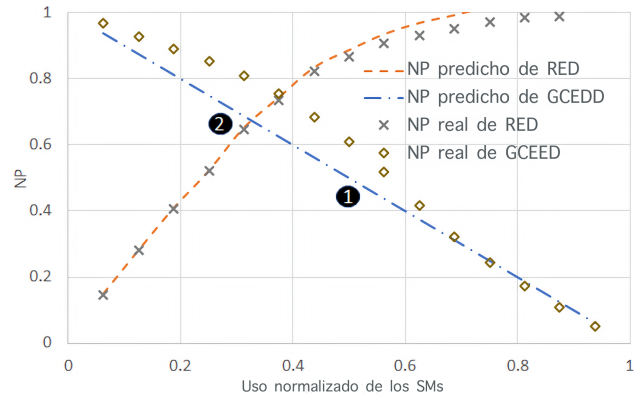
$$IPC_{CB}^{alone} = IPC_{CB}^{shared} \times \frac{R_{CB}^{alone}}{R_{CB}^{shared}} \quad (5.2)$$

$$NP = \frac{R_{CB}^{shared}}{R_{CB}^{alone}} \quad (5.3)$$

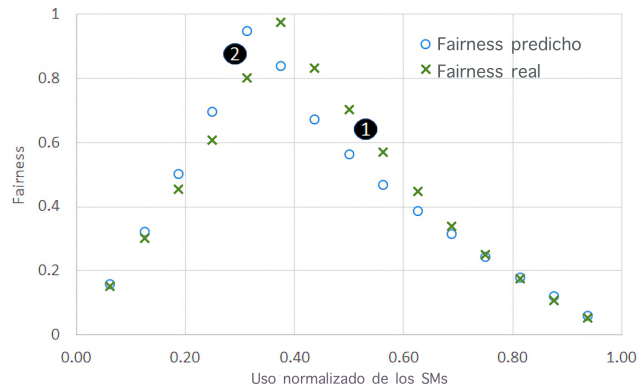
Los valores de los NP s predichos puede utilizarse para elegir una planificación justa de los SMs durante la coejecución de *kernels*. El *fairness* (Ecuación 5.4) puede definirse como la relación entre el NP de los *kernels* i y j [25], donde el valor calculado oscila entre 0 (no *fairness*) y 1 (*fairness* perfecto, los *kernels* obtienen una ejecución equilibrada). HSM-Fair comienza asignando el mismo número de bloques de cada *kernel* sobre los SMs, prediciendo el NP y el *fairness*. Entonces, se calcula cuantos bloques deberían lanzarse del *kernel* con mayor NP y del *kernel* con menor NP para obtener un NP similar para ambos *kernels*. Este procedimiento se repite hasta que el *fairness* predicho es superior a un umbral (por ejemplo, 0,9).

$$Fairness = \min_{i,j} \left(\frac{NP_i}{NP_j} \right) \quad (5.4)$$

Las figuras 5.4a y 5.4b muestran un ejemplo de la aplicación de HSM-Fair a la coejecución de los *kernels* RED y GCEDD aplicando SMK. La figura 5.4a muestra los valores predichos y reales de NP para ambos *kernels*. Los valores del eje de abscisas representan el porcentaje de CTAs lanzados por RED, con respecto al número máximo de CTAs persistentes (16 para RED). Así, un valor del 50 % significa una configuración con 8 CTAs lanzados por RED, y el mayor número de CTAs posibles de GCEDD. RED es un *kernel* limitado por la memoria y casi no hay error en la predicción. GCEDD es un *kernel* de cómputo y su NP no tiene un comportamiento lineal. La figura 5.4b muestra el *fairness* predicho y real para cada planificación. El *fairness* máximo predicho se encuentra a la izquierda del *fairness* máximo real. HSM-Fair comienza con una distribución del 50 %, anotada con un círculo negro y el número 1 en la figura. Los valores NP predichos (\widehat{NP}) y el NP real, junto al *fairness* predicho ($\widehat{Fairness}$) y el real se muestran en la tabla 5.1. El error en la predicción se acerca al 20 % pero, sin embargo, HSM-Fair asume correctamente que se deben lanzar más CTAs del *kernel* de cómputo. A continuación, se prueba la configuración del 30 %, que supone lanzar aproximadamente un 70 % de los CTAs del *kernel* de cómputo. HSM-Fair predice un *fairness* de 0,949, que está por encima del umbral del 0,9, por lo que deja de buscar una configuración mejor. Por desgracia, el error de predicción es muy alto y el *fairness* real es de 0,8. Si el umbral fuera más alto y se probara la mejor configuración real, la del 40 %, HSM-Fair predeciría un *fairness* inferior a la del 30 % y fijaría de nuevo esta planificación.



(a)



(b)

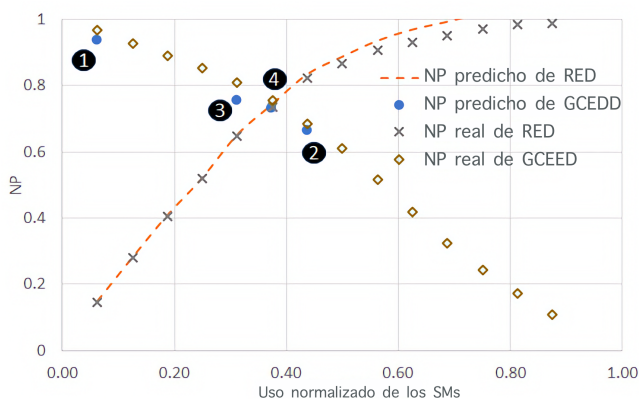
Figura 5.4: (a) NP y (b) Fairness para HSM-Fair aplicado a la coejecución SMK de RED y GCEED

HSM				
#	\widehat{NP}	NP	$\widehat{Fairness}$	$Fairness$
1	0.5	0.611	0.563	0.703
2	0.687	0.810	0.949	0.800

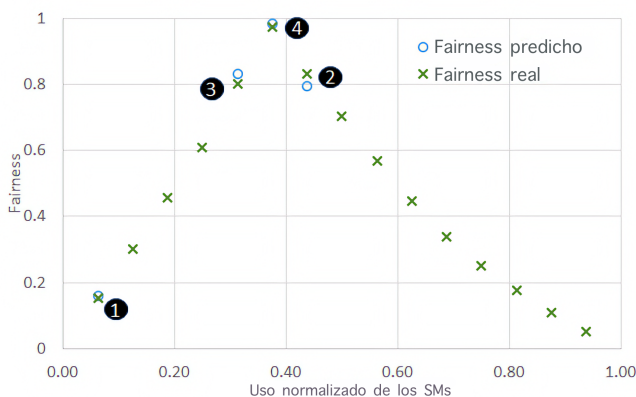
Tabla 5.1: Valores HSM para las figuras 5.4a y 5.4b.

HSM puede usarse para predecir correctamente el NP de los *kernels* limitados por memoria, pero falla para los *kernels* de cómputo. Desgraciadamente,

las suposiciones sobre el comportamiento lineal en el rendimiento a medida que cambia la asignación de recursos, que funcionan bien para SMT, no se sostiene para SMK. Una de las principales razones es que la ecuación 5.2 no logra predecir el IPC^{alone} utilizando el IPC^{shared} con una configuración del 50%. En esta tesis proponemos un modelo híbrido de *slowdown* por partes (HPSM, *Hybrid Piecewise Slowdown Model*) que predice el IPC^{alone} utilizando el IPC^{shared} en la planificación con más CTAs. Este modelo predice cuantos CTAs deben asignarse a un *kernel* de cómputo utilizando intervalos de planificación ya probados.



(a)



(b)

Figura 5.5: (a) NP y (b) Fairness para HPSM-Fair aplicado a la coejecución SMK de RED y GCEED

Las figuras 5.5a y 5.5b muestran un ejemplo de la aplicación del modelo HPSM

para la coejecución de los *kernels* RED y GCEDD utilizando una planificación SMK. La figura 5.5a muestra los valores predichos y reales para NP del *kernel* limitado por la memoria, RED. Para GCEDD, solo muestra los valores probados por este método, ya que la predicción para una planificación depende de las configuraciones ya probadas. La figura 5.5b muestra el *fairness* real para todas las planificaciones y los valores predichos para las configuraciones probadas. Este modelo comienza con la asignación de un solo CTA para RED y quince CTAs para GCEDD. El IPC_1^{shared} en este punto se usa para predecir $\widehat{IPC}^{alone} = \frac{16}{15} \times IPC_1^{shared}$, así $\widehat{NP}_1 = \frac{15}{16}$. La tabla 5.2 muestra los valores predichos y reales de NP y *Fairness* y, como se puede observar, el error de la predicción es muy bajo. Ahora, el intervalo de predicción está entre el IPC_1^{shared} y 0 (el valor del IPC con 0 CTAs asignados). HPSM-Fair usa este intervalo para predecir una planificación con nueve CTAs por SM para GCEDD. HSM-Fair habría predicho $\widehat{NP}_2 = 0,562$ pero nuestro modelo corrige la predicción usando $\widehat{NP}_2 = \frac{IPC_2^{shared}}{IPC^{alone}}$ que es cercano al valor real. Ahora, hay dos intervalos, uno entre IPC_1^{shared} y IPC_2^{shared} y otro intervalo entre IPC_2^{shared} y 0. HPSM-Fair predice la asignación de más CTAs para el *kernel* de cómputo, usando el primer intervalo para predecir una tercera planificación al 30%, coincidiendo con la segunda planificación testeada usando HSM-Fair. Ahora, el error es mucho menor y HPSM-Fair predice correctamente para probar una cuarta configuración, al 40%, donde encuentra la configuración óptima para el *fairness*.

#	HPSM			
	\widehat{NP}	NP	$\widehat{Fairness}$	<i>Fairness</i>
1	0.937	0.967	0.160	0.152
2	0.662	0.683	0.793	0.832
3	0.754	0.810	0.830	0.800
4	0.732	0.755	0.983	0.975

Tabla 5.2: Valores HPSM para las figuras 5.5a y 5.5b.

5.4. Implementación *hardware*

Para implementar las estrategias SMT y SMK hemos modificado el planificador de CTAs del simulador GPGPU-sim en su versión 4.0 [12], y le hemos añadido soporte para la ejecución concurrente de *kernels*.

```

1 while (true) {
2   id = getNextSM ();

```

```

3  if ( notMoreCTAsLeft( currentKernel ) )
4      currentKernel = getNextKernel();
5  if ( enoughResources( id , currentKernel ) )
6      issueCTAtoSM( id , currentKernel);
7  }

```

Código 5.1: Planificador de CTAs

El planificador de CTAs del simulador (ver código 5.1) intenta lanzar un CTA sobre un SM durante cada ciclo de planificación. Para ello dispone de un par de punteros, `currentKernel` e `id`, que almacenan la información sobre el *kernel* que se está ejecutando y el identificador del SM *core* en el que se lanzará un CTA, y una serie de funciones auxiliares para implementar toda la lógica necesaria:

- `getNextSM()` devuelve el identificador del siguiente SM sobre el que se intentará lanzar un CTA. Esta función sigue un esquema de tipo *round robin* para recorrer todos los SMs.
- `notMoreCTAsLeft(currentKernel)` comprueba si quedan más CTAs por lanzar del *kernel* actual.
- `getNextKernel()` devuelve el identificador del siguiente *kernel* preparado para ser ejecutado.
- `enoughResources(id, currentKernel)` comprueba si el SM con identificador `id` tiene suficientes recursos para ejecutar un CTA de `currentKernel`.
- `issueCTAtoSM(id, currentKernel)` lanza un CTA de `currentKernel` sobre el SM con identificador `id`.

Con este planificador solo se pueden ejecutar concurrentemente dos *kernels* cuando el primero ha terminado de lanzar todos sus CTAs y queda espacio para lanzar los primeros CTAs del segundo *kernel*. No obstante, es relativamente simple incorporar soporte para incluir la estrategia SMT: basta con añadir un registro, `SMTrange`, que almacene el rango de SMs donde lanzaremos CTAs del primer *kernel*, y dos punteros `currentKernel1` y `currentKernel2` para almacenar la información sobre los dos *kernels* que se quieren coejecutar. El código 5.2 muestra la lógica que sigue el planificador SMT: si el identificador del próximo SM se encuentra entre 0 y `SMTrange - 1` se lanza un CTA del primer *kernel* y en caso contrario del segundo *kernel*. En el caso de que se quiera generalizar para coejecutar más de 2 *kernels* bastaría con añadir otro registro por cada *kernel* adicional.


```

1 while( true ) {
2     id = getNextSM();
3     if ( id < SMTrange ) {
4         if ( notMoreCTAsLeft(currentKernel1) )
5             currentKernel1 = getNextKernel();
6         if ( enoughResources(id, currentKernel1) )
7             issueCTAtoSM(id, currentKernel1);
8     } else {
9         if ( notMoreCTAsLeft(currentKernel2) )
10            currentKernel2 = getNextKernel();
11        if ( enoughResources(id, currentKernel2) )
12            issueCTAtoSM(id, currentKernel2);
13    }
14 }

```

Código 5.2: Planificador SMT de CTAs

La implementación de la estrategia SMK es algo más compleja ya que se necesitan dos registros, `SMKLimitK1` y `SMKLimitK2`, para anotar el número máximo de CTAs de cada *kernel* que se pueden lanzar en cada SM, y otros dos registros por cada SM *core* que almacenen el número actual de CTAs de cada *kernel* lanzados sobre ese SM. Además, hay que modificar la función `issueCTAtoSM()` para que devuelva el número de CTAs de ese *kernel* que se están ejecutando sobre ese SM. El código 5.3 muestra el funcionamiento del planificador. Durante cada ciclo de planificación se intenta lanzar un CTA de cada *kernel*; para ello deben quedar suficientes recursos en el SM, y no debe haberse alcanzado el límite de CTAs que tiene asignado ese *kernel*.

```

1 while( true ) {
2     id = getNextSM();
3     if ( notMoreCTAsLeft(currentKernel1) )
4         currentKernel1 = getNextKernel();
5     if ( notMoreCTAsLeft(currentKernel2) )
6         currentKernel2 = getNextKernel();
7     if ( SMKcounterK1 < SMKlimitK1 &
8         enoughResources(id, currentKernel1) )
9         SMKcounterK1 = issueCTAtoSM(id, currentKernel1);
10    if ( SMKcounterK2 < SMKlimitK2 &
11        enoughResources(id, currentKernel2) )
12        SMKcounterK2 = issueCTAtoSM(id, currentKernel2);
13 }

```

Código 5.3: Planificador SMK de CTAs

Para implementar las políticas HSM-Fair y HPSM-Fair basta con comprobar periódicamente los valores de *IPC*, ancho de banda y *RBH*, y actualizar los valores de `SMTrange` para el caso de SMT, o `SMKLimitK1` y `SMKLimitK2` para el caso de SMK. Al igual que en [107] la comprobación se hace sobre ventanas

de 500.000 ciclos de duración para obtener valores estables de cada uno de los parámetros.

5.5. Resultados experimentales

Para validar nuestro modelo hemos utilizado la versión modificada de GPGPU-Sim descrita en la sección previa con una arquitectura Volta equivalente a la de una GPU Titan V. Dicha GPU modela 40 *clusters* de 2 SM *cores* cada uno, sumando un total de 80 SMs, conectados a una memoria HBM de 3 *stacks* y 24 canales.

Los experimentos se han realizando utilizando diferentes *kernels* pertenecientes a las *suites* de CUDA SDK [66], Rodinia [18], Parboil [89] y Chai [27]. La tabla 5.3 muestra los nombres de los *kernels*, el número máximo de CTAs persistentes por *cluster* y la categoría en la que se clasifican utilizando el procedimiento explicado en [107]. Se han emparejado todos los *kernels* limitados por memoria (MB) con los *kernels* limitados por cómputo (CB), obteniendo 48 pares de *kernels*. Estos se han utilizado para comparar la planificación SMK con la planificación SMT y el rendimiento obtenido para una planificación *fairness* utilizando nuestro modelo de *slowdown* híbrido (HPSM). Se ha simulado cada combinación lanzando cada *kernel* al mismo tiempo y parando cuando uno de ellos ha terminado su ejecución.

Las métricas utilizadas son las definidas por [25] para medir el rendimiento. Estas métricas son Normalized Progress (*NP*), Average Normalized Turnaround Time (*ANTT*), System Throughput (*STP*) y Fairness. NP_i viene dado por la ecuación 5.5, donde IPC_i^{shared} es el *IPC* alcanzado por el *kernel* i cuando se coejecuta con otro *kernel*, mientras que IPC_i^{alone} es el *IPC* del *kernel* i se ejecuta en solitario. La inversa de *NP* es el *Normalized turnaround time* (*NTT*) y puede utilizarse para definir el *ANTT* (ecuación 5.6) e indica el *slowdown* dado por la ejecución concurrente de los *kernels* i y j .

$$NP_i = \frac{IPC_i^{shared}}{IPC_i^{alone}} \quad (5.5)$$

$$ANTT = \frac{1}{2} \left(\frac{1}{NP_i} + \frac{1}{NP_j} \right) \quad (5.6)$$

Cuanto menor es el valor de *ANTT*, mejor es el resultado. *STP* cuantifica el progreso acumulado de los *kernels* i y j durante la ejecución concurrente y se

Kernel	Aplicación	CTAs por cluster	Categoría
HIST	Histogram [66]	16	MB
BS	Black Scholes [66]	32	MB
VA	Vector Addition [66]	16	MB
RED	Reduction [66]	16	MB
RCONV	Rows convolution [66]	64	MB
SRAD	Diffusion algorithm [18]	16	MB
LBM	Lattice-Boltzmann method [89]	20	MB
EULER	Euler3D [18]	10	MB
GCEDD	Gaussian Canny edge detection [27]	16	CB
PF	Pathfinder [18]	16	CB
MM	Matrix Multiplication [66]	16	CB
HS	Hotspot [18]	10	CB
DXTC	DXT Compression [66]	24	CB
BOPTS	Binomial options [66]	32	CB

Tabla 5.3: *Kernels* usados en los experimentos para la validación del modelo *hardware*.

calcula utilizando la ecuación 5.1, cuanto mayor sea su valor, mejor es el resultado. Finalmente, hemos utilizado la métrica de *fairness* definida por la ecuación 5.4.

Todos los *kernels* se han ejecutado en solitario para obtener su IPC^{alone} . También se ha medido el ratio de aciertos en el *Row Buffer* (RBH) para cada *kernel* y el uso del ancho de banda para calcular el modelo lineal que se utiliza con HSM y HPSM.

5.5.1. SMT vs SMK

Las combinaciones de *kernels* se han simulado utilizando todas las configuraciones posibles de SMT y SMK. En SMT, cada *kernel* se ejecuta con un número de *cluster* de SMs entre 1 y 39 y el otro *kernel* con el número restante de *clusters* para completar los 40 *clusters* que tiene esta arquitectura. En SMK, a cada *kernel* se le asigna entre 1 y el número máximo de CTAs persistentes por *cluster* (mostrado en la columna 3 de la tabla 5.3), mientras que al otro *kernel* se le asignan el número de CTAs que sea posible para completar el *cluster*. A continuación, se ha calculado los valores de IPC^{shared} para todas las combinaciones y se han calculado el *ANTT* y *STP*. Finalmente, se han seleccionado los mejores resultados para cada combinación tanto para SMT, como para SMK. En las figuras 5.6a y 5.6b se muestran los *boxplots* de estos resultados con el fin de

mostrar sus valores estadísticos [93]. En cada caja, la marca central corresponde a la mediana, mientras que los bordes de la caja representa los percentiles 25 y 75, y los bigotes se extienden hasta los valores más extremos no considerados como valores atípicos (alrededor de $\pm 2,7\sigma$ y un 99,3% de cobertura). SMK supera claramente a SMT tanto en *ANTT* como en *STP*. Esto se debe principalmente al hecho de que, en la planificación SMT, los CTAs del mismo *kernel* compiten por los mismos recursos dentro de un SM. Con la planificación SMK hay CTAs de *kernels* diferentes dentro de un mismo SM, que no tienen porque competir por los mismos recursos, lo que reduce la interferencia entre los *kernels* durante la coejecución de los mismos.

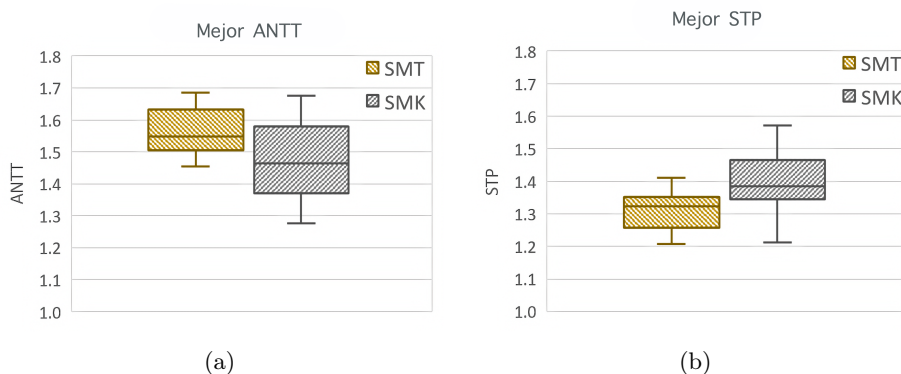


Figura 5.6: *Boxplots* de los mejores resultados para (a) ANTT y (b) STP conseguidos durante la coejecución de *kernels* para SMT y SMK.

5.5.2. Política *Fairness*

En esta tesis, se ha propuesto un modelo *hardware* denominado HPSM con el fin de predecir el *NP* de los *kernels* que se ejecutan de forma concurrente. HPSM modela los *kernels* limitados por la memoria utilizando el mismo esquema que HSM y usa una predicción lineal de *NP* para los *kernels* limitados por cómputo, pero añade un nivel de aproximación por tramos para reducir los errores de predicción. Estas predicciones pueden utilizarse para asignar CTAs de forma que se cumplan ciertos criterios, como el *fairness*, pero los errores al realizar la predicción pueden llevar a planificaciones erróneas que no cumplan con dichos criterios. El error en la predicción se puede medir como un porcentaje de error relativo, tal y como se muestra en la ecuación 5.7, donde $\hat{\alpha}$ es el valor predicho y α es el

valor real.

$$100 \times \frac{|\hat{\alpha} - \alpha|}{\alpha} \quad (5.7)$$

En este trabajo hemos comparado el rendimiento alcanzado por nuestro modelo HPSM-Fair con el obtenido por el modelo HSM-Fair cuando se usa la estrategia de asignación SMK.

En primer lugar hemos comparado los errores en la predicción cuando se calculan los valores de *fairness*, *ANTT* y *STP*. Las gráficas de la columna izquierda de la figura 5.7 muestran los *boxplots* de los errores cometidos por las predicciones realizadas. La altura de las cajas para la predicción de los errores para HSM-Fair es mayor porque el *NP* de algunos *kernels* limitados por el cómputo no muestran un comportamiento lineal y los errores pueden ser muy variados. Por otro lado, la distribución de los errores de HPSM-Fair muestra una variabilidad mucho menor ya que la aproximación por tramos se adapta mejor al comportamiento no lineal. Además, la mediana de los errores de predicción de HPSM-Fair es menor para cualquiera de las tres métricas. Concretamente, la mediana para el *fairness* está por debajo del 10 % mientras que HSM-Fair comete un error cercano al 30 %. Para *ANTT* y *STP* los errores de nuestro modelo están en torno al 5 % mientras que los errores de HSM-Fair varían entre el 11 % y el 17 %.

Las gráficas de la columna derecha de la figura 5.7 muestran los valores reales de *fairness*, *ANTT* y *STP* obtenidos con HSM y HPSM, junto con los valores con la mejor planificación real en *fairness*. Los valores se han ordenado de menor a mayor para mostrar más claramente las diferencias, así que los valores situados en la misma columna pueden corresponderse a parejas de *kernels* distintas. HSM no consigue un *fairness* óptimo en la mayoría de combinaciones de *kernels*, lo que conduce a un *STP* más bajo y a un *ANTT* más alto para algunas combinaciones de *kernels*. Por el contrario, HPSM selecciona planificaciones casi óptimas respecto al *fairness* la mayoría de las veces y los valores de *ANTT* y *STP* se acercan más a los valores con una planificación óptima en *fairness*. En la gráfica que muestra el valor de *STP* alcanzado por la configuración seleccionada por cada método se puede ver que nuestro método obtiene valores superiores a los alcanzados por el óptimo. Esto ocurre en algunos casos en los que nuestro método selecciona una pareja que no es óptima en términos de *fairness* pero alcanza un valor de *STP* superior a la que sí es óptima respecto al *fairness*.

Estos resultados tienen un coste mayor dado a que la convergencia para encontrar la mejor planificación requiere de más tiempo. Se han medido el número de planificaciones usadas para encontrar la mejor distribución por cada méto-

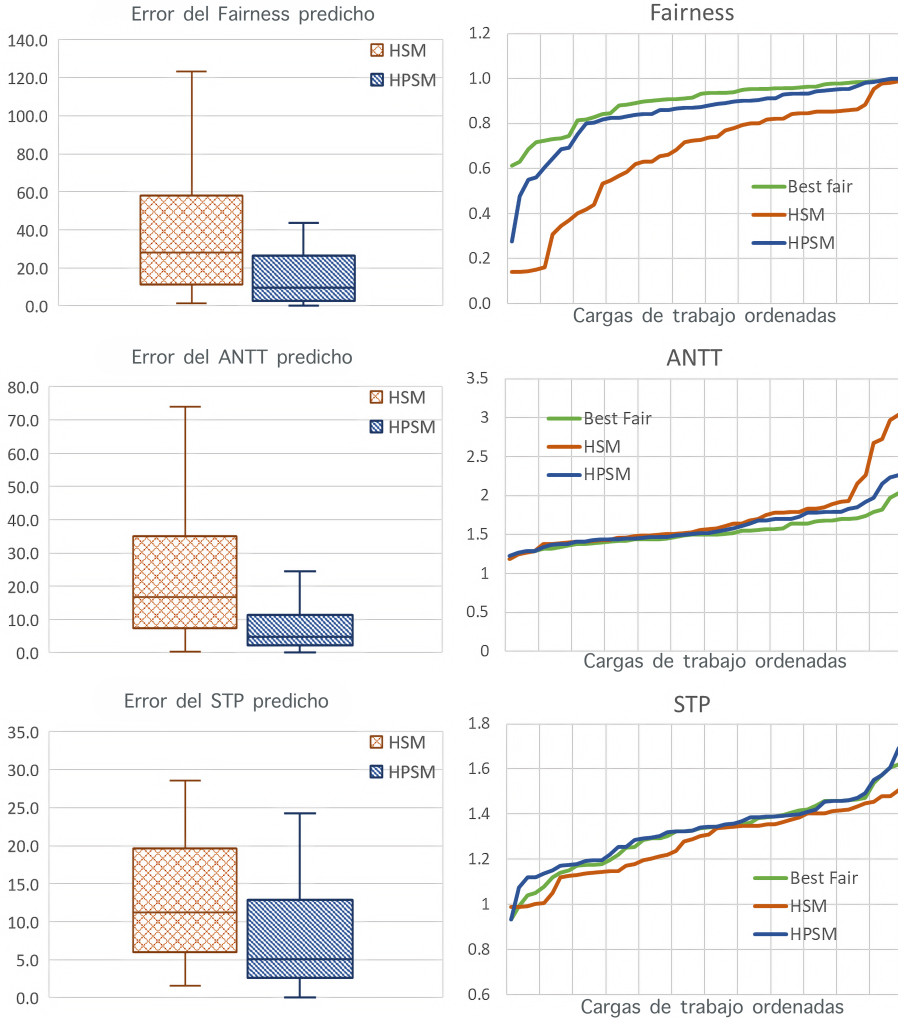


Figura 5.7: Columna izquierda: *boxplots* del error en la predicción del *Fairness*, *ANTT* y *STP*. Columna derecha: resultados obtenidos con una política *fairness* de planificación de CTAs para las combinaciones ejecutadas.

do: HSM prueba 2,9 planificaciones de media, mientras que HPSM llega hasta 4,2 porque empieza a probar planificaciones en un extremo, mientras que HSM empieza en el centro.

Por último, hemos comparado los resultados obtenidos por HPSM-Fair y HSM-Fair cuando se utiliza una estrategia SMT. Los resultados son prácticamente iguales, con errores medianos del 5,21% en ambos métodos. En este caso HPSM-Fair necesita una media de 2,21 planificaciones mientras que HSM-Fair obtiene su predicción en 2,15 planificaciones.

5.6. Resumen

En este capítulo hemos presentado HPSM (*Hybrid Piecewise Slowdown Model*), un modelo *hardware* de planificación y ejecución concurrente de *kernels* (CKE) en una GPU. Este modelo se ha desarrollado con el objetivo de mejorar el tiempo de ejecución de un conjunto de *kernels* y que, además, pueda ser utilizado para aplicar políticas orientadas al *fairness*. La propuesta realizada se ha hecho usando el simulador GPGPU-Sim, uno de los simuladores *hardware* de GPU más avanzados en la actualidad.

Nuestra implementación *hardware* requiere de un número reducido de cambios en la arquitectura de la GPU y es capaz de asignar CTAs a los SMs usando dos estrategias distintas: SMT y SMK. Además, hemos desarrollado un modelo que permite obtener predicciones del progreso normalizado de los *kernels* para las distintas configuraciones de coejecución usando cualquiera de las dos estrategias. Estas predicciones nos permiten diseñar un planificador capaz de aplicar políticas orientadas al *fairness* al que hemos denominado HPSM-Fair.

Hemos realizado una serie de experimentos usando un conjunto de 48 parejas de *kernels*, donde uno de ellos está limitado por memoria y el otro limitado por cómputo, para comparar las estrategias de asignación de CTAs y validar nuestro modelo. En primer lugar, hemos probado todas las configuraciones de coejecución para cada una de las dos estrategias y hemos seleccionado los mejores valores de STP y ANTT. La estrategia SMK supera claramente a SMT en ambas métricas, tal y como se puede comprobar en la Figura 5.6.

A continuación hemos comparado nuestro modelo con el estado del arte para obtener una planificación basada en el *fairness* de parejas de *kernels* usando la estrategia SMK. Por un lado hemos comparado el error de predicción de ambos modelos sobre las métricas de *fairness*, STP y ANTT. En todos los casos nuestro modelo obtiene predicciones mucho más precisas, con un error mediano tanto de STP como ANTT en torno al 5%, y menor del 10% para el *fairness*. En contraposición, los errores de predicción de HSM son bastante mayores, con un error mediano superior al doble alcanzado por HPSM para todas las métricas.

Por otro lado, hemos comparado las configuraciones seleccionadas por HSM-Fair con las que obtiene nuestro modelo, HPSM-Fair. Nuevamente nuestro modelo supera claramente al estado del arte, obteniendo configuraciones muy próximas al óptimo, aunque con el inconveniente de converger más lentamente ya que debe probar una media de 1,3 planificaciones más que HSM-Fair.

6 Conclusiones

En esta tesis se ha analizado el problema de planificar un conjunto de tareas sobre una GPU desde diferentes puntos de vista. Por una parte se ha estudiado el solapamiento de comandos de transferencia de datos con comandos de ejecución de *kernels* con el objetivo de minimizar el tiempo de ejecución (*makespan*). Por otra parte se han comparado distintos métodos que permiten la ejecución solapada de varios *kernels* sobre la misma GPU buscando alcanzar diferentes objetivos como maximizar el rendimiento del sistema (*system throughput*), alcanzar la equidad (*fairness*) o garantizar una calidad de servicio (QoS).

En este capítulo se presentan, en la sección 6.1, las contribuciones y conclusiones de esta tesis. A continuación, en la sección 6.2 se enumeran los artículos publicados con los resultados de esta tesis y, por último, en la sección 6.3 se plantean algunas posibles líneas futuras de investigación.

6.1. Contribuciones y conclusiones

Para abordar el estudio de la superposición de comandos se ha presentado en el Capítulo 3 un modelo que simula la ejecución de múltiples tareas en una GPU. Este modelo puede ser empleado para identificar el orden de ejecución que resulte en un tiempo de procesamiento mínimo. En contraposición con anteriores investigaciones que examinan las estrategias de planificación de tareas en GPUs, en este trabajo se considera la teoría de planificación en general para alcanzar el objetivo planteado. En la presente tesis se aplican los conceptos de dicha teoría a este problema, demostrando que la ejecución concurrente de tareas en una GPU a través de *streams* de CUDA se puede modelar como un problema de tipo *Flow Shop*. La

principal ventaja de este enfoque radica en que se puede definir una función objetivo y seleccionar una solución adecuada a partir de la literatura existente. Como ejemplo práctico, se ha analizado el problema $F3|prmu|C_{max}$, que surge cuando varios hilos lanzan *kernels* independientes que pueden ser procesados utilizando el mismo contexto GPU.

En el estado del arte existen diversas soluciones para la planificación de tareas en GPUs, como por ejemplo las heurísticas SI, NEH y SQ. En esta tesis se ha desarrollado una nueva estrategia llamada NEH-GPU, que combina una heurística previamente existente (NEH) con un modelo de ejecución de tareas en GPU. Esta estrategia también es utilizable en tiempo de ejecución debido a su bajo *overhead* y a que no requiere modificación de los *kernels* originales. El modelo de ejecución de tareas en GPU incluye un modelo preciso de transferencia de datos que calcula el tiempo de ejecución de cada comando de transferencia de datos. Este modelo es compatible con diferentes arquitecturas de GPU (Kepler, Maxwell y Pascal), tipos de memoria (paginable y *pinned*) y transferencias de datos simultáneas.

Se han llevado a cabo varios experimentos para validar la eficacia y robustez del modelo. Se han evaluado tres arquitecturas de GPU distintas: Kepler, Maxwell y Pascal, empleando una variedad de *kernels* reales de los SDK de CUDA y Rodinia. Se ha efectuado un análisis estadístico para evaluar la relevancia de la heurística NEH-GPU y mostrar su superioridad en comparación con otras heurísticas. Además, se ha analizado el efecto al variar la cantidad de tareas con el fin de evaluar la escalabilidad de las heurísticas. En todos los experimentos NEH-GPU logró resultados iguales o superiores al mejor *makespan* alcanzado por cualquier heurística, y muy cercanos al óptimo. Por último, se realizó una comparación con MPS que demuestra que nuestro planificador ofrece aceleraciones que oscilan entre 1,15 y 1,20 sobre la solución de Nvidia.

En esta tesis también se ha abordado el problema de la ejecución concurrente de *kernels* (CKE). En este tipo de problemas se busca planificar un conjunto de *kernels* para su coejecución, mediante una distribución adecuada de los bloques de los distintos *kernels* sobre los Streaming Multiprocessors (SMs), que permita mejorar el uso de los recursos *hardware* y alcanzar algún objetivo de planificación. Hemos analizado este problema tanto desde el punto de vista *software* (Capítulo 4) como *hardware* (Capítulo 5).

Desde el punto de vista *software* hemos desarrollado un modelo para la ejecución concurrente de *kernels* en una GPU. Este modelo, denominado *FlexSched*, implementa políticas de planificación destinadas a maximizar el rendimiento en la ejecución de los *kernels* o a satisfacer requisitos de calidad de servicio (QoS) de la misma, como por ejemplo el tiempo máximo de respuesta de un *kernel*. Una

ventaja importante de *FlexSched* es que requiere solo modificaciones mínimas en el código del *kernel* para lograr una distribución eficiente de los recursos de la GPU.

Además, *FlexSched* incorpora un módulo que planifica la ejecución de los *kernels* y utiliza datos recolectados en una fase de *profiling* productivo para optimizar la distribución de recursos. Una de las principales características que distingue a *FlexSched* de otros planificadores es su uso de un *profiler on-line* productivo. Durante la fase de *profiling*, la heurística del planificador compara diferentes configuraciones de ejecución para buscar un esquema de planificación óptimo que cumpla con los requisitos de rendimiento o de QoS. Gracias a esta flexibilidad en la planificación, *FlexSched* puede ser aplicado a situaciones reales en las que *kernels* desconocidos deben ser ejecutados de forma inmediata en la GPU, a diferencia de otros trabajos previos que requieren un costoso *profiling off-line*.

En el proceso de validación de *FlexSched* se han llevado a cabo numerosos experimentos utilizando 9 aplicaciones con un total de 13 *kernels*. Primero, para comparar nuestra propuesta con otros enfoques del estado del arte para la coejecución de *kernels*, se comparó rtSMK, nuestro planificador de CTAs desarrollado para *FlexSched*, con un método de *slicing* denominado cCUDA, logrando valores de *STP* entre 1.1 y 2.1 y una mejora promedio del 10% en comparación con cCUDA. También se realizó un experimento que demostró que el aumento en el *STP* logrado por la coejecución de los *kernels* depende en gran medida de elegir una planificación adecuada. Además, se demostró que la heurística utilizada por el planificador *FlexSched* es capaz de encontrar la planificación que alcanza el mayor rendimiento en la mayoría de los casos, con una pérdida promedio de rendimiento con respecto a la mejor planificación posible inferior al 1%.

FlexSched también se comparó con el planificador hardware usado por HyperQ, mejorando el rendimiento de los *kernels* en un factor de 1.22. Finalmente, se realizaron experimentos con aplicaciones sensibles a la latencia, en los cuales *FlexSched* calculó cuántos recursos de la GPU podrían asignarse al resto de aplicaciones sin afectar los requisitos de la aplicación sensible a la latencia. Una vez más, se demostró la importancia del proceso de *profiling* para aprovechar el uso de los recursos de la GPU mientras se cumplen las restricciones de QoS.

En esta tesis también hemos desarrollado HPSM (*Hybrid Piecewise Slowdown Model*), un modelo *hardware* de planificación y ejecución concurrente de *kernels* en una GPU. Este modelo se ha desarrollado con el objetivo de mejorar el tiempo de ejecución de un conjunto de *kernels* y que, además, pueda ser utilizado para aplicar políticas orientadas al *fairness*. La propuesta realizada se ha hecho usando

el simulador GPGPU-Sim, uno de los simuladores *hardware* de GPU más avanzados en la actualidad. La implementación *hardware* de nuestra solución requiere pocos cambios en la arquitectura de la GPU y es capaz de asignar CTAs a los SMs usando dos técnicas diferentes: SMT y SMK. También hemos añadido un modelo que puede predecir el progreso normalizado de los *kernels* para diferentes configuraciones de coejecución utilizando cualquiera de estas dos técnicas. Con estas predicciones, hemos desarrollado un planificador llamado HPSM-Fair que se enfoca en aplicar políticas de *fairness*.

Para validar el modelo realizamos experimentos con un conjunto de 48 parejas de *kernels*, uno de los *kernels* limitado por memoria y el otro por cómputo, y evaluamos las estrategias de asignación de CTAs. En primer lugar probamos todas las configuraciones de coejecución para cada estrategia y seleccionamos los mejores valores de STP y ANTT, llegando a la conclusión de que la estrategia SMK supera significativamente a SMT en ambas métricas. Luego, comparamos nuestro modelo, HPSM-Fair, con el estado del arte para lograr una planificación de *kernels* basada en el *fairness*. Comparamos el error de predicción de ambos modelos en las métricas de *fairness*, STP y ANTT, y encontramos que HPSM tiene predicciones más precisas, con un error mediano tanto de STP como ANTT en torno al 5%, y menor al 10% para el *fairness*. Por otro lado, comparamos las configuraciones seleccionadas por HPSM-Fair con las de HSM-Fair, y descubrimos que HPSM supera claramente al estado del arte, obteniendo configuraciones cercanas al óptimo, pero con una convergencia más lenta, ya que debe probar una media de 1.3 planificaciones adicionales.

En resumen, las propuestas de *FlexSched* y HPSM representan un avance en la planificación y ejecución concurrente de *kernels* en una GPU que combinan diferentes factores para mejorar el tiempo de ejecución y aplicar políticas de QoS y *fairness*. Los experimentos realizados demuestran la efectividad de las propuestas y su potencial para ser utilizadas en diferentes contextos.

6.2. Aportaciones

Las aportaciones presentadas en esta tesis han dado lugar a una serie de publicaciones que son listadas a continuación en orden cronológico:

- A.J. Lázaro-Muñoz, J.M. González-Linares, B. López-Albelda, N. Guil, A Scheduling Theory Framework for GPU Tasks Efficient Execution. Publicado en VECPAR 2018.

Aplicación de la teoría de planificación en la ejecución concurrente de tareas en una GPU para reducir el tiempo de computación de una carga de trabajo. En este trabajo se construye un modelo que tiene en cuenta las capacidades *hardware* del acelerador, las características de la carga de trabajo, las limitaciones y las funciones objetivo siguiendo un esquema similar al utilizado en la teoría de planificación. En nuestro modelo, la planificación de tareas en GPU es modelada como un problema de planificación *Flow Shop*, que nos permite aplicar y comparar métodos ya conocidos en las operaciones de búsqueda. Además, desarrollamos una nueva heurística, centrada específicamente en la ejecución de comandos en GPU, que consigue mejores resultados de planificación que otras técnicas ya conocidas.

- B. López-Albelda, J.M. Gonzalez-Linares, N. Guil, Tasks Fairness Scheduler for GPU. Jornadas de Paralelismo. Sarteco, 2019.

En este trabajo, presentamos un mecanismo de *preemption* por *software* muy eficaz y con poca sobrecarga en la ejecución original de los *kernels*, el cual permite desalojar y volver a lanzar los *kernels* de la GPU para dar soporte a políticas de planificación distintas. Además, proponemos un planificador justo denominado *Fair and Responsive Scheduler* (FRS), que tiene en cuenta el *slowdown* producido en los *kernels* con el fin de seleccionar el *kernel* a lanzar y establecer el tiempo de ejecución mediante un *quantum*. Este planificador se compara con algunos de los planificadores más usados como *Shortest Job First Scheduler* (SJF), *Shortest Remaining Time Scheduler* (SRT), *Round Robin Scheduler* (RR) y *Completely Fair Scheduler* (CFS). La comparativa con el resto de planificadores se ha realizado con las métricas: *Average Normalized Tournaroung Time* (ANTT), *Deviation Normalized Tournaroun Time* (DNTT) o *System Overall Throughput* (STP). La métrica DNTT muestra que FRS obtiene planificaciones más justas que el resto de planificadores, 1,5 veces mejor que SRT, el segundo planificador más justo.

- B. López-Albelda, A.J. Lázaro-Muñoz, J.M. González-Linares, N. Guil, Heuristics for concurrent task scheduling on GPUs. Publicado en *Concurrency and Computation: Practice and Experience* en 2020.

En este artículo ampliamos el trabajo presentado en el congreso *13th International Meeting on High Performance Computing for Computational Science* (VECPAR 2018). Partiendo de la heurística desarrollada para resolver el problema de *Flow Shop* con el que modelamos la planificación de tareas sobre una GPU, hicimos una evaluación más exhaustiva del modelo que demostraron la idoneidad y la solidez de este nuevo enfoque. Además, se

añadió una comparación con MPS (*Multi-Process Service*), la API de NVIDIA que se ocupa de la ejecución de tareas concurrentes, que nos permitió comprobar que nuestra solución obtiene tiempos de ejecución menores, con factores de aceleración que oscilan entre 1,15 y 1,20.

- B. López-Albelda, F.M. Castro, J.M. Gonzalez-Linares, N. Guil, FlexSched: Efficient scheduling techniques for concurrent kernel execution on GPUs. Publicado en The Journal of Supercomputing en 2022.

En este trabajo presentamos un planificador *software*, denominado *FlexSched*, que emplea un mecanismo en tiempo de ejecución con una baja sobrecarga que realiza la asignación de bloques de hilos siguiendo una distribución *intra-SM* de los *kernels* que se coejecutan (SMK). También implementa un mecanismo productivo de *profiling* en línea que permite cambiar dinámicamente la asignación de recursos de los *kernels* atendiendo al rendimiento instantáneo alcanzado por los *kernels* coejecutados. Una característica importante de nuestro enfoque es que no se requiere un análisis del *kernel* fuera de línea para establecer la mejor asignación de recursos de los *kernels* coejecutados. Por lo tanto, puede funcionar en cualquier sistema en el que deban planificarse nuevas aplicaciones de forma inmediata. Utilizando un conjunto de 9 aplicaciones (13 *kernels* distintos), demostramos que nuestro enfoque mejora el rendimiento de la ejecución concurrente alcanzado por los métodos de planificación más recientes. Además, probamos *FlexSched* en un escenario de planificación real en el que las nuevas aplicaciones se lanzan tan pronto como los recursos de la GPU están disponibles. En este escenario, *FlexSched* reduce el tiempo medio de ejecución global en un factor de 1,25 veces con respecto al tiempo obtenido cuando se emplea el *hardware* propio de Nvidia, HyperQ. Por último, usamos *FlexSched* para implementar políticas de planificación que garanticen un tiempo de respuesta máximo para las aplicaciones sensibles a la latencia, a la vez que se consigue un alto uso de los recursos mediante la coejecución de *kernels*.

- B. López-Albelda, F.M. Castro, J.M. Gonzalez-Linares, N. Guil, A hybrid piece-wise slowdown model for concurrent kernel execution on GPU. Publicado en European Conference on Parallel and Distributed Computing (Hybrid) en 2022.

En este artículo se estudia la aplicación de mecanismos de planificación concurrente de *kernels* por medio de métodos *hardware* que permiten mejorar el uso de los recursos de la GPU y reducir el tiempo de ejecución de los *kernels* coejecutados. Además, se implementan políticas de planificación eficientes que persiguen criterios basados en la equidad. La mejora de la ejecución concurrente depende en gran medida de la forma en que se reparten

los recursos de la GPU entre los *kernels*. Por tanto, es necesario utilizar modelos de predicción del *slowdown* precisos que predigan el rendimiento de la coejecución con exactitud para cumplir los requisitos de la política de planificación. La mayoría de los modelos de predicción del *slowdown* recientes trabajan con planificaciones *Spatial Multitasking* (SMT), pero en este trabajo mostramos que el particionamiento de tipo *Simultaneous Multikernel* (SMK) obtiene una mejor productividad. Sin embargo, la interferencia de los *kernels* asignados con SMK se produce no sólo en la memoria global, como en el caso del SMT, sino también dentro del SM, lo que provoca elevados errores de en la predicción del *slowdown*. En este trabajo proponemos una modificación de un modelo de predicción del *slowdown* para reducir el error de predicción medio del 27,92% al 9,50%. Además, este nuevo modelo de predicción del *slowdown* se utiliza para implementar una política de planificación que mejora la equidad en 1,41x de media en comparación con el particionamiento uniforme, mientras que los modelos anteriores sólo alcanzan 1,21x de media.

6.3. Líneas futuras de investigación

En estas tesis se ha estudiado la planificación de tareas y la ejecución de *kernels* en las GPUs y han surgido algunos aspectos que podrían constituir nuevas líneas de investigación.

6.3.1. Ejecución concurrente de *kernels* y transferencias de memoria

Las líneas de investigación principales presentadas en esta tesis, se han basado en: 1) la ejecución concurrente de comandos de aplicaciones, donde los comandos de transferencia se pueden solapar entre sí y junto a la ejecución de *kernels*; y 2) la ejecución concurrente de dos *kernels* complementarios. Esta segunda línea de investigación requiere que los *kernels* estén preparados para su ejecución y, por lo tanto, las transferencias de memoria necesarias se han realizado con anterioridad.

Un nuevo campo de estudio podría ser la planificación simultánea de comandos donde, además de permitir el solapamiento de comandos de transferencia de datos con *kernels*, se permita la ejecución concurrente de *kernels* que estén listos para su ejecución. Este problema se podría modelar mediante un *Flow Shop* flexible donde la etapa de procesamiento en la GPU se descompondría en varias etapas

en paralelo para simular la coexistencia de varios *kernels* sobre la GPU.

6.3.2. Aplicación de modelos predictivos

En el estudio de la ejecución concurrente de comandos se ha utilizado un modelo lineal de predicción del tiempo de ejecución de los *kernels* en la GPU. Este modelo es muy simple, precisa de una fase previa de *profiling* y no es exacto, por lo que la planificación propuesta puede ser inadecuada y, por tanto, no obtener el mejor orden de lanzamiento de los comandos en una GPU. Además, la fase de *profiling* requiere de un tiempo de ejecución previo que puede ser improductivo.

En trabajos futuros se pueden desarrollar modelos más sofisticados para la predicción del rendimiento usando por ejemplo técnicas de aprendizaje automático que puedan reducir el error del modelo lineal actual. También se podrían aplicar modelos predictivos que mediante la recolección de datos de los contadores *hardware* internos permitan identificar patrones en el código que nos permitan predecir una buena distribución de recursos *hardware* en la coejecución de *kernels*.

6.3.3. Procesamiento en memoria

En esta tesis se han planteado mecanismos y arquitecturas *hardware* basadas en GPUs actuales cuyos *cores* poseen las mismas características. El principal cuello de botella en estas arquitecturas reside en el acceso a la memoria global ya que, en muchas ocasiones, no hay reaprovechamiento de los datos. Una forma de solucionar este problema es el uso de arquitecturas de procesamiento en memoria (PIM) en la GPU, y en estas arquitecturas podrían aplicarse técnicas de CKE para ejecutar *kernels* intensivos en memoria mientras se ejecutan los *kernels* intensivos en cómputo en los *cores* normales de la GPU.

Bibliografía

- [1] ADRIAENS, J. T., COMPTON, K., KIM, N. S., AND SCHULTE, M. J. The case for gpgpu spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture* (2012), IEEE, pp. 1–12. (Cited on page 116)
- [2] AHN, J., HONG, S., YOO, S., MUTLU, O., AND CHOI, K. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), pp. 105–117. (Cited on page 1)
- [3] AHN, J., YOO, S., MUTLU, O., AND CHOI, K. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)* (2015), IEEE, pp. 336–348. (Cited on page 1)
- [4] AL FAISAL, F., AND RAHMAN, M. H. Symmetric tori connected torus network. In *2009 12th International Conference on Computers and Information Technology* (2009), IEEE, pp. 174–179. (Cited on page 47)
- [5] ALEXANDROV, A., IONESCU, M. F., SCHAUSER, K. E., AND SCHEIMAN, C. Loggp: Incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures* (1995), pp. 95–105. (Cited on pages 53 and 61)
- [6] AMD. Amd gpus. <https://www.amd.com/es/graphics/radeon-rx-graphics>, 2021. (Cited on page 1)
- [7] AMD. Compute cores. https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf, Enero 2014. (Cited on page 11)

- [8] APPUSWAMY, R., FRANCESCHETTI, M., KARAMCHANDANI, N., AND ZEGER, K. Network computing capacity for the reverse butterfly network. In *2009 IEEE International Symposium on Information Theory (2009)*, IEEE, pp. 259–262. (Cited on page 47)
- [9] ASCNOW. Asc american sun components. <https://www.ascnow.com/>, Junio 2021. (Cited on page 1)
- [10] AUGONNET, C., THIBAUT, S., NAMYST, R., AND WACRENIER, P.-A. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198. (Cited on page 63)
- [11] AVNET-ASIC. Asic design services. <https://www.avnet-asic.com/>, Junio 2021. (Cited on page 1)
- [12] BAKHODA, A., YUAN, G. L., FUNG, W. W., WONG, H., AND AAMODT, T. M. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software (2009)*, IEEE, pp. 163–174. (Cited on pages 8, 9, 44 and 127)
- [13] BASARAN, C., AND KANG, K.-D. Supporting preemptive task executions and memory copies in gpgpus. In *2012 24th Euromicro Conference on Real-Time Systems (2012)*, IEEE, pp. 287–296. (Cited on page 55)
- [14] BOYER, M., MENG, J., AND KUMARAN, K. Improving gpu performance prediction with data transfer modeling. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (2013)*, IEEE, pp. 1097–1106. (Cited on page 53)
- [15] C. C. programming languages. <https://www.cprogramming.com//>, Julio 2021. (Cited on page 29)
- [16] CARVALHO, P., CRUZ, R., DRUMMOND, L., BENTES, C., CLUA, E., CATALDO, E., AND MARZULO, L. A. Kernel concurrency opportunities based on gpu benchmarks characterization. *Cluster Computing* 23, 1 (2020), 177–188. (Cited on page 99)
- [17] CHATTERJEE, N., O’CONNOR, M., LOH, G. H., JAYASENA, N., AND BALASUBRAMONIA, R. Managing dram latency divergence in irregular gpgpu applications. In *SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2014)*, IEEE, pp. 128–139. (Cited on page 83)

- [18] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC) (2009)*, Ieee, pp. 44–54. (Cited on pages 44, 71, 98, 130 and 131)
- [19] CHEN, G., ZHAO, Y., SHEN, X., AND ZHOU, H. Effisha: A software framework for enabling efficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2017)*, pp. 3–16. (Cited on pages 82, 92 and 101)
- [20] CHEN, Q., YANG, H., GUO, M., KANNAN, R. S., MARS, J., AND TANG, L. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (2017)*, pp. 17–32. (Cited on pages 85, 92 and 94)
- [21] CHI, P., LI, S., XU, C., ZHANG, T., ZHAO, J., LIU, Y., WANG, Y., AND XIE, Y. Prime: A novel processing-in-memory architecture for neural network computation in rram-based main memory. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 27–39. (Cited on page 1)
- [22] CULLER, D. E., KARP, R. M., PATTERSON, D., SAHAY, A., SANTOS, E. E., SCHAUSER, K. E., SUBRAMONIAN, R., AND VON EICKEN, T. Logp: A practical model of parallel computation. *Communications of the ACM* 39, 11 (1996), 78–85. (Cited on pages 53 and 61)
- [23] DAI, H., LIN, Z., LI, C., ZHAO, C., WANG, F., ZHENG, N., AND ZHOU, H. Accelerate gpu concurrent kernel execution by mitigating memory pipeline stalls. In *2018 IEEE international symposium on high performance computer architecture (HPCA) (2018)*, IEEE, pp. 208–220. (Cited on page 117)
- [24] DURAN, A., AYGUADÉ, E., BADIA, R. M., LABARTA, J., MARTINELL, L., MARTORELL, X., AND PLANAS, J. Ompps: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters* 21, 02 (2011), 173–193. (Cited on page 63)
- [25] EYERMAN, S., AND EECKHOUT, L. System-level performance metrics for multiprogram workloads. *IEEE micro* 28, 3 (2008), 42–53. (Cited on pages 124 and 130)

- [26] GARCÍA, L. P., CUENCA, J., AND CÁNOVAS, D. G. On optimization techniques for the matrix multiplication on hybrid cpu+ gpu platforms. *Annals of Multicore and GPU Programming: AMGP 1*, 1 (2014), 1–8. (Cited on page 53)
- [27] GÓMEZ-LUNA, J., EL HAJJ, I., CHANG, L.-W., GARCÍA-FLORESZX, V., DE GONZALO, S. G., JABLIN, T. B., PENA, A. J., AND HWU, W.-M. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2017), IEEE, pp. 43–54. (Cited on pages 98, 130 and 131)
- [28] GÓMEZ-LUNA, J., GONZÁLEZ-LINARES, J. M., BENAVIDES, J. I., AND GUIL, N. Performance models for asynchronous data transfers on consumer graphics processing units. *Journal of Parallel and Distributed Computing* 72, 9 (2012), 1117–1126. (Cited on page 55)
- [29] GRAHAM, R. L., LAWLER, E. L., LENSTRA, J. K., AND KAN, A. R. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Annals of discrete mathematics*, vol. 5. Elsevier, 1979, pp. 287–326. (Cited on page 64)
- [30] HENNESSY, J. L., AND PATTERSON, D. A. *Computer architecture: a quantitative approach*. Elsevier, 2011. (Cited on page 122)
- [31] HSIEH, K., EBRAHIMI, E., KIM, G., CHATTERJEE, N., O’CONNOR, M., VIJAYKUMAR, N., MUTLU, O., AND KECKLER, S. W. Transparent of-flooding and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. In *ACM SIGARCH Computer Architecture News* (2016), IEEE Press, pp. 204–216. (Cited on page 83)
- [32] HU, Q., SHU, J., FAN, J., AND LU, Y. Run-time performance estimation and fairness-oriented scheduling policy for concurrent gpgpu applications. In *2016 45th International Conference on Parallel Processing (ICPP)* (2016), IEEE, pp. 57–66. (Cited on page 5)
- [33] HUANG, J.-C., LEE, J. H., KIM, H., AND LEE, H.-H. S. Gpumech: Gpu performance modeling technique based on interval analysis. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), IEEE, pp. 268–279. (Cited on page 43)
- [34] INTEL. 7th generation intel coreprocessor, intel celeron processor, and intel xeon processor e3 v6 family desktop, workstation, and mobile platforms.

- <https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/7th-gen-core-family-datasheet-addendum.pdf>, Abril 2017. (Cited on page 11)
- [35] JIANG, N., MICHELOGIANNAKIS, G., BECKER, D., TOWLES, B., AND DALLY, W. Booksim interconnection network simulator. *Online*, <https://nocs.stanford.edu/cgi-bin/trac.cgi/wiki/Resources/BookSim> (2016). (Cited on page 46)
- [36] JOG, A., KAYIRAN, O., CHIDAMBARAM NACHIAPPAN, N., MISHRA, A. K., KANDEMIR, M. T., MUTLU, O., IYER, R., AND DAS, C. R. Owl: cooperative thread array aware scheduling techniques for improving gpgpu pformance. *ACM SIGPLAN Notices* 48, 4 (2013), 395–406. (Cited on page 83)
- [37] JOG, A., KAYIRAN, O., KESTEN, T., PATRNAIK, A., BOLOTIN, E., CHATTERJEE, N., KECKLER, S. W., KANDEMIR, M. T., AND DAS, C. R. Anatomy of gpu memory system for multi-application execution. In *Proceedings of the 2015 International Symposium on Memory Systems* (2015), ACM, pp. 223–234. (Cited on page 83)
- [38] JOHNSON, S. M. Optimal two-and three-stage production schedules with setup times included. *Naval research logistics quarterly* 1, 1 (1954), 61–68. (Cited on page 67)
- [39] KARAMI, A., MIRSOLEIMANI, S. A., AND KHUNJUSH, F. A statistical performance prediction model for opencl kernels on nvidia gpus. In *The 17th CSI International Symposium on Computer Architecture & Digital Systems (CADS 2013)* (2013), IEEE, pp. 15–22. (Cited on page 54)
- [40] KATO, S., LAKSHMANAN, K., RAJKUMAR, R., ISHIKAWA, Y., ET AL. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)* (2011). (Cited on page 82)
- [41] KHAIRY, M., SHEN, Z., AAMODT, T. M., AND ROGERS, T. G. Accel-sim: An extensible simulation framework for validated gpu modeling. In *47th IEEE/ACM International Symposium on Computer Architecture (ISCA)* (2020), IEEE. (Cited on page 118)
- [42] KHRONOS GROUP. Opencl 2.0 api specification. <https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf>, Octubre 2014. (Cited on pages 2, 11 and 28)

- [43] KONSTANTINIDIS, E., AND COTRONIS, Y. A practical performance model for compute and memory bound gpu kernels. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2015), IEEE, pp. 651–658. (Cited on page 54)
- [44] LÁZARO-MUÑOZ, A., GONZÁLEZ-LINARES, J. M., GÓMEZ-LUNA, J., AND GUIL, N. A tasks reordering model to reduce transfers overhead on gpus. *Journal of Parallel and Distributed Computing* 109 (2017), 258–271. (Cited on pages 61, 62, 63, 66, 68, 69 and 82)
- [45] LEE, H., AND AL FARUQUE, M. A. Gpu-evr: Run-time event based real-time scheduling framework on gpgpu platform. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2014), IEEE, pp. 1–6. (Cited on pages 82, 84, 95 and 98)
- [46] LEE, M., SONG, S., MOON, J., KIM, J., SEO, W., CHO, Y., AND RYU, S. Improving gpgpu resource utilization through alternative thread block scheduling. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (2014), IEEE, pp. 260–271. (Cited on pages 43 and 82)
- [47] LEMEIRE, J., CORNELIS, J. G., AND SEGERS, L. Microbenchmarks for gpu characteristics: The occupancy roofline and the pipeline model. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)* (2016), IEEE, pp. 456–463. (Cited on page 54)
- [48] LEYS, C., LEY, C., KLEIN, O., BERNARD, P., AND LICATA, L. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of experimental social psychology* 49, 4 (2013), 764–766. (Cited on page 77)
- [49] LI, Z., FANG, J., TANG, T., CHEN, X., CHEN, C., AND YANG, C. Evaluating the performance impact of multiple streams on the mic-based heterogeneous platform. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2016), IEEE, pp. 1341–1350. (Cited on page 55)
- [50] LIANG, Y., HUYNH, H. P., RUPNOW, K., GOH, R. S. M., AND CHEN, D. Efficient gpu spatial-temporal multitasking. *IEEE Transactions on Parallel and Distributed Systems* 26, 3 (2014), 748–760. (Cited on pages 82, 84, 93 and 94)

- [51] LIN, Z., NYLAND, L., AND ZHOU, H. Enabling efficient preemption for simt architectures with lightweight context switching. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2016), IEEE, pp. 898–908. (Cited on page 116)
- [52] LIU, B., QIU, W., JIANG, L., AND GONG, Z. Software pipelining for graphic processing unit acceleration: Partition, scheduling and granularity. *The International Journal of High Performance Computing Applications* 30, 2 (2016), 169–185. (Cited on pages 55 and 63)
- [53] LIU, Y., YU, Z., EECKHOUT, L., REDDI, V. J., LUO, Y., WANG, X., WANG, Z., AND XU, C. Barrier-aware warp scheduling for throughput processors. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), pp. 1–12. (Cited on page 43)
- [54] MARGIOLAS, C., AND O'BOYLE, M. F. Portable and transparent host-device communication optimization for gpgpu environments. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2014), pp. 55–65. (Cited on page 58)
- [55] MENG, J., MOROZOV, V. A., KUMARAN, K., VISHWANATH, V., AND URAM, T. D. Grophecy: Gpu performance projection from cpu code skeletons. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), IEEE, pp. 1–11. (Cited on page 54)
- [56] NAWAZ, M., ENSCORE JR, E. E., AND HAM, I. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega* 11, 1 (1983), 91–95. (Cited on page 68)
- [57] NVIDIA. CUPTI: User guide. Version: DA-05679-001_v11e.1, October 2020. (Cited on pages 36 and 99)
- [58] NVIDIA. Nvidia gpus. <https://www.nvidia.com/es-es/studio/compare-gpus/>, 2021. (Cited on pages 1 and 43)
- [59] NVIDIA. Nvidia multi-instance gpu. *Online*, <https://www.nvidia.com/en-us/technologies/multi-instance-gpu/> (2022). (Cited on page 27)
- [60] NVIDIA. Cuda multiprocess-service. https://docs.nvidia.com/pdf/CUDA_Multi_Process_Service_Overview.pdf, Abril 2022. (Cited on page 67)

- [61] NVIDIA. nsight. <https://developer.nvidia.com/nsight-visual-studio-edition>, Agosto 2021. (Cited on page 36)
- [62] NVIDIA. Nvidia visual profiler. <https://developer.nvidia.com/nvidia-visual-profiler>, Agosto 2021. (Cited on page 36)
- [63] NVIDIA. nvprof. <https://docs.nvidia.com/cuda/profiler-users-guide/index.html>, Agosto 2021. (Cited on page 36)
- [64] NVIDIA. nvprof profiling options. <https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-command-line-options>, Agosto 2021. (Cited on page 36)
- [65] NVIDIA. Nvidia tegra k1. a new era in mobile computing. https://www.nvidia.com/content/PDF/tegra_white_papers/tegra-K1-whitepaper.pdf, Enero 2014. (Cited on page 11)
- [66] NVIDIA. CUDA Samples, version 10.1.105, February 2019. (Cited on pages 4, 71, 98, 130 and 131)
- [67] NVIDIA. Kepler gk110/210. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf>, Junio 2021. (Cited on page 17)
- [68] NVIDIA. Nvidia ampere ga102 gpu architecture. <https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf>, Junio 2021. (Cited on page 27)
- [69] NVIDIA. Nvidia geforce gtx 980. <https://www.techpowerup.com/gpu-specs/docs/nvidia-gtx-980.pdf>, Junio 2021. (Cited on page 21)
- [70] NVIDIA. Nvidia tesla p100 (pascal architecture). <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf>, Junio 2021. (Cited on page 22)
- [71] NVIDIA. Nvidia tesla v100 gpu architecture (volta architecture). <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>, Junio 2021. (Cited on page 24)
- [72] NVIDIA. Nvidia turing gpu architecture. <https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/>

- turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, Junio 2021. (Cited on page 26)
- [73] NVIDIA. Nvlink and nvswitch. <https://www.nvidia.com/es-es/data-center/nvlink/>, Junio 2021. (Cited on page 16)
- [74] NVIDIA. Cuda programming guide. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>, Septiembre 2015. (Cited on pages 2, 11, 28, 54, 55, 56 and 58)
- [75] PAI, S., GOVINDARAJAN, R., AND THAZHUTHAVEETIL, M. J. Preemptive thread block scheduling with online structural runtime prediction for concurrent gpgpu kernels. In *Proceedings of the 23rd international conference on Parallel architectures and compilation* (2014), pp. 483–484. (Cited on page 82)
- [76] PAI, S., THAZHUTHAVEETIL, M. J., AND GOVINDARAJAN, R. Improving gpgpu concurrency with elastic kernels. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 407–418. (Cited on pages 82, 83 and 98)
- [77] PALMER, D. Sequencing jobs through a multi-stage process in the minimum total time—a quick method of obtaining a near optimum. *Journal of the Operational Research Society* 16, 1 (1965), 101–107. (Cited on page 67)
- [78] PARK, J. J. K., PARK, Y., AND MAHLKE, S. Chimera: Collaborative preemption for multitasking on a shared gpu. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 593–606. (Cited on pages 116 and 117)
- [79] PARK, J. J. K., PARK, Y., AND MAHLKE, S. Dynamic resource management for efficient utilization of multitasking gpus. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems* (2017), pp. 527–540. (Cited on pages 117 and 121)
- [80] PATTNAIK, A., TANG, X., JOG, A., KAYIRAN, O., MISHRA, A. K., KANDEMIR, M. T., MUTLU, O., AND DAS, C. R. Scheduling techniques for gpu architectures with processing-in-memory capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (2016), ACM, pp. 31–44. (Cited on page 83)
- [81] PEKHIMENKO, G., BOLOTIN, E., VIJAYKUMAR, N., MUTLU, O., MOWRY, T. C., AND KECKLER, S. W. A case for toggle-aware compression for gpu systems. In *2016 IEEE International Symposium on High Performance*

- Computer Architecture (HPCA)* (2016), IEEE, pp. 188–200. (Cited on page 83)
- [82] PINEDO, M., AND HADAVI, K. Scheduling: theory, algorithms and systems development. In *Operations research proceedings 1991*. Springer, 1992, pp. 35–42. (Cited on page 65)
- [83] ROGERS, T. G., OCONNOR, M., AND AAMODT, T. M. Cache-conscious wavefront scheduling. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), IEEE, pp. 72–83. (Cited on page 43)
- [84] RUIZ, R., AND MAROTO, C. A comprehensive review and evaluation of permutation flowshop heuristics. *European journal of operational research* 165, 2 (2005), 479–494. (Cited on page 67)
- [85] SHEKOFTEH, S.-K., NOORI, H., NAGHIBZADEH, M., FRÖNING, H., AND YAZDI, H. S. ccuda: Effective co-scheduling of concurrent kernels on gpus. *IEEE Transactions on Parallel and Distributed Systems* 31, 4 (2019), 766–778. (Cited on pages 83, 99, 102 and 114)
- [86] SIM, J., DASGUPTA, A., KIM, H., AND VUDUC, R. A performance analysis framework for identifying potential benefits in gpgpu applications. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (2012), pp. 11–22. (Cited on page 54)
- [87] SOUZA, P., NEWBURN, C., AND BORGES, L. Heterogeneous architecture library. In *Second EAGE Workshop on High Performance Computing for Upstream* (2015), European Association of Geoscientists & Engineers, pp. 1–5. (Cited on page 55)
- [88] STANISIC, L., THIBAUT, S., LEGRAND, A., VIDEAU, B., AND MÉHAUT, J.-F. Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures. *Concurrency and Computation: Practice and Experience* 27, 16 (2015), 4075–4090. (Cited on page 54)
- [89] STRATTON, J. A., RODRIGUES, C., SUNG, I.-J., OBEID, N., CHANG, L.-W., ANSSARI, N., LIU, G. D., AND HWU, W.-M. W. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 29. (Cited on pages 130 and 131)
- [90] SUPERCOMPUTER, T. Top500 supercomputer list. <https://www.top500.org/>, 2021. (Cited on page 1)

- [91] TAILLARD, E. Some efficient heuristic methods for the flow shop sequencing problem. *European journal of Operational research* 47, 1 (1990), 65–74. (Cited on page 68)
- [92] TANASIC, I., GELADO, I., CABEZAS, J., RAMIREZ, A., NAVARRO, N., AND VALERO, M. Enabling preemptive multiprogramming on gpus. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 193–204. (Cited on page 116)
- [93] TUKEY, J. W., ET AL. *Exploratory data analysis*, vol. 2. Reading, MA, 1977. (Cited on pages 72 and 132)
- [94] VAN WERKHOVEN, B., MAASSEN, J., SEINSTRAS, F. J., AND BAL, H. E. Performance models for cpu-gpu data transfers. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2014), IEEE, pp. 11–20. (Cited on pages 53, 54, 56, 60 and 62)
- [95] VIJAYKUMAR, N., PEKHIMENKO, G., JOG, A., BHOWMICK, A., AUSAVARUNGNIRUN, R., DAS, C., KANDEMIR, M., MOWRY, T. C., AND MUTLU, O. A case for core-assisted bottleneck acceleration in gpus: enabling flexible data compression with assist warps. In *ACM SIGARCH Computer Architecture News* (2015), ACM, pp. 41–53. (Cited on page 83)
- [96] WAGNER, H. M. An integer linear-programming model for machine scheduling. *Naval research logistics quarterly* 6, 2 (1959), 131–140. (Cited on page 67)
- [97] WANG, H., LUO, F., IBRAHIM, M., KAYIRAN, O., AND JOG, A. Efficient and fair multi-programming in gpus via effective bandwidth management. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018), IEEE, pp. 247–258. (Cited on page 117)
- [98] WANG, Z., YANG, J., MELHEM, R., CHILDERS, B., ZHANG, Y., AND GUO, M. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), IEEE, pp. 358–369. (Cited on pages 82, 95, 117 and 121)
- [99] WENDE, F., CORDES, F., AND STEINKE, T. On improving the performance of multi-threaded cuda applications with concurrent kernel execution by kernel reordering. In *2012 Symposium on Application Accelerators in High Performance Computing* (2012), IEEE, pp. 74–83. (Cited on page 82)

- [100] WU, B., CHEN, G., LI, D., SHEN, X., AND VETTER, J. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (2015), pp. 119–130. (Cited on pages 82, 84, 90, 93 and 110)
- [101] WU, B., LIU, X., ZHOU, X., AND JIANG, C. Flep: Enabling flexible and efficient preemption on gpus. *ACM SIGPLAN Notices* 52, 4 (2017), 483–496. (Cited on pages 84, 92 and 94)
- [102] XILINX. Xilinx fpgas. <https://www.xilinx.com/products/silicon-devices/fpga.html/>, 2021. (Cited on page 1)
- [103] XU, Q., JEON, H., KIM, K., RO, W. W., AND ANNAVARAM, M. Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (2016), IEEE, pp. 230–242. (Cited on pages 5, 85, 116, 118 and 121)
- [104] YU, C., BAI, Y., YANG, H., CHENG, K., GU, Y., LUAN, Z., AND QIAN, D. Smguard: A flexible and fine-grained resource management framework for gpus. *IEEE Transactions on Parallel and Distributed Systems* 29, 12 (2018), 2849–2862. (Cited on pages 82, 84, 98, 110 and 112)
- [105] ZHAO, C., GAO, W., NIE, F., AND ZHOU, H. A survey of gpu multitasking methods supported by hardware architecture. *IEEE Transactions on Parallel and Distributed Systems* 33, 6 (2021), 1451–1463. (Cited on page 116)
- [106] ZHAO, W., CHEN, Q., LIN, H., ZHANG, J., LENG, J., LI, C., ZHENG, W., LI, L., AND GUO, M. Themis: Predicting and reining in application-level slowdown on spatial multitasking gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2019), IEEE, pp. 653–663. (Cited on pages 82, 117 and 121)
- [107] ZHAO, X., JAHRE, M., AND EECKHOUT, L. Hsm: A hybrid slowdown model for multitasking gpus. In *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems* (2020), pp. 1371–1385. (Cited on pages 118, 119, 121, 122, 129 and 130)
- [108] ZHENG, T., NELLANS, D., ZULFIQAR, A., STEPHENSON, M., AND KECKLER, S. W. Towards high performance paged memory for gpus. In *2016*

- IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), IEEE, pp. 345–357. (Cited on page 59)
- [109] ZHONG, J., AND HE, B. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2013), 1522–1532. (Cited on pages 82, 83, 98 and 102)
- [110] ZIABARI, A. K., SUN, Y., MA, Y., SCHAA, D., ABELLÁN, J. L., UBAL, R., KIM, J., JOSHI, A., AND KAELI, D. Umh: A hardware-based unified memory hierarchy for systems with multiple discrete gpus. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 4 (2016), 1–25. (Cited on page 54)



UNIVERSIDAD
DE MÁLAGA