UNIVERSIDAD
DE MÁLAGA

Escuela Técnica Superior de
Ingeniería Informática

Programa de Doctorado en
Ingenería Mecatrónica

TESIS DOCTORAL

Planificación Dinámica de Tareas en Aceleradores

Antonio José Lázaro Muñoz

Directores:
Nicolás Guil Mata
Jose María González Linares
Juan Gómez Luna

2019

UNIVERSIDAD DE MÁLAGA

AUTOR: Antonio José Lázaro Muñoz

http://orcid.org/0000-0002-1867-413X

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga

UNIVERSIDAD
DE MÁLAGA

Departamento de Arquitectura de Computadores

Programa de Doctorado
Ingeniería Mecatrónica

TESIS DOCTORAL

Planificación Dinámica de Tareas en Aceleradores

Antonio José Lázaro Muñoz

Enero de 2019

Dirigida por:
Nicolás Guil Mata
José Mª González Linares
Juan Gómez Luna

Dr. D. Nicolás Guil Mata.
Catedrático del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

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

Dr. D. Juan Gómez Luna.
Profesor Asociado del Departamento de Arquitectura de Computadores y Electrónica de la Universidad de Córdoba.

CERTIFICAN:

Que la memoria titulada "Planificación Dinámica de Tareas en Aceleradores", ha sido realizada por D. Antonio José Lázaro Muñoz 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 Informática.

Málaga, Enero de 2019

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

Dr. D. José Mª González Linares.
Codirector de la tesis.

Dr. D. Juan Gómez Luna.
Codirector de la tesis.

A mis padres y hermano

Índice general

Contenido I

Lista de Figuras V

Lista de Tablas XV

1.- Introducción **1**

1.1. Motivación. Ejemplo 2

1.2. Entornos de Desarrollo 5

1.2.1. Puerto PCI Express (Peripheral Component Interconnect Express) 6

1.2.2. Arquitecturas 7

Aceleradores NVIDIA 7

Aceleradores AMD 16

Aceleradores Intel 18

1.2.3. Interfaces de Programación (APIs) 20

NVIDIA-CUDA 20

OpenCL 31

1.3. Objetivos de la Tesis 45

1.4. Estructura de la Tesis 46

2.- Ejecución Concurrente de Tareas **49**

2.1. Estado del Arte 50

2.1.1. Modelado de Transferencias 50

2.1.2. Modelado de Kernels 52

2.1.3. Ejecución Concurrente de Comandos 52

2.2. Lanzamiento Asíncrono de Comandos 54

2.2.1. CUDA Streams 55

2.2.2. Colas de Comandos en OpenCL 56

2.3. Estimación de Tiempos de Ejecución de los Comandos 59

2.3.1. Reserva y Liberación de Memoria 60

2.3.2. Transferencias de Memoria 61

2.3.3. Kernels 65

2.4. Modelo General. Colas de Simulación 65

2.5. Benchmarks 69

2.5.1. Sintéticos 69

2.5.2. Reales 73

2.6. Validación del Modelo 73

2.7. Resumen 76

3.- Planificador Dinámico **81**

3.1. Estado del arte 81

3.2. Planificador Dinámico 84

3.3. Definición de los Comandos 86

3.4. Heurística de Planificación 92

3.5. Validación 94

3.5.1. Resultados para Entornos NVIDIA-CUDA 95

3.5.2. Resultados para Entornos OpenCL 105

3.6. Resumen 112

4.- Teoría de Planificación **117**

4.1. Teoría de Planificación 117

4.2. Búsqueda de una Planificación Óptima 121

4.2.1. Heurística de la Pendiente 122

4.2.2. Heurística de Planificación NEH 123

4.2.3. Heurística de Planificación NEH-GPU 124

4.3. Validación 126

4.3.1. Análisis Estadístico 127

4.3.2. Escalabilidad 134

4.4. Resumen 136

5.- Conclusiones **139**

5.1. Contribuciones y conclusiones 139

5.2. Aportaciones 141

5.3. Líneas futuras de investigación 143

5.3.1. Restricciones de memoria en el acelerador 143

5.3.2. Aplicación de la Teoría de Planificación 144

5.3.3. Ejecución concurrente de kernels 144

Bibliografía **147**

Índice de figuras

1.1. Dos ejemplos de ejecución concurrente de las mismas cuatro tareas, pertenecientes al SDK de CUDA, sobre un acelerador NVIDIA K20c. Las tareas Matrix Multiplication (MM-Kernel), Black Scholes (BS-Kernel), Separable Convolution (SC-Kernel) y Vector Addition (V) son lanzadas por los streams 0, 1, 2 y 3, respectivamente. Las transferencias *host to device* (HTD), los comandos de *kernels* y las transferencias *device to host* (DTH) son representadas mediante cajas azules, verdes y amarillas, respectivamente. Cada uno de los ejemplos muestran un orden de lanzamiento de tareas diferente. 3

1.2. Arquitectura Tesla. Esta arquitectura está compuesta por un *Texture Processor Cluster* (TPC). Cada TPC contiene 2 SMs (*Streaming Multiprocessors*) que están compuestos a su vez por 8 SPs (*Streaming Processors*) y 2 SFUs (*Special Functions Units*). . . . 9

1.3. Diseño de un SM (*Streaming Multiprocessor*) en la arquitectura Fermi. Esta arquitectura está compuesta por 16 SMs que están compuestos a su vez por 32 SPs (*Streaming Processors*), 16 unidades LD/ST (Load and Store Units) y 4 SFUs (*Special Functions Units*). Fermi presenta un doble planificador de *warps* para ejecutar concurrentemente dos *warps* en el mismo SM. 9

1.4. El diseño de un SM es mejorado en Kepler dando lugar al diseño denominado SMX (*Next Generation Streaming Multiprocessor*). Esta arquitectura está compuesta por 15 SMs que están compuestos a su vez por 192 SPs (*Streaming Processors*), 32 unidades LD/ST (Load and Store Units) y 32 SFUs (*Special Functions Units*). Kepler es capaz de ejecutar 4 *warps* en un SMX, debido a su cuádruple planificador de *warps*. 10

1.5. En el modelo de Fermi mostrado a la izquierda, solamente las parejas de kernels (C, P) y (R, X) pueden ejecutarse concurrentemente debido a la dependencia causada por el uso de una sola cola hardware. El modelo de Kepler utiliza Hyper-Q, que permite a todos los streams ejecutarse concurrentemente usando colas hardware separadas. 12

1.6. Flujo de trabajo entre el *host* y la GPU para Fermi (izquierda) y para Kepler (derecha). El rediseño del flujo de trabajo en Kepler muestra una nueva unidad llamada GMU (Grid Management Unit), que permite lanzar, pausar y supender las *grids* activas. . . 13

1.7. El diseño de un SM en Maxwell GM204 (diseño adoptado también por la arquitectura Pascal). El número de SPs es reducido de 192 en Kepler a 128 en Maxwell. El número de SPs son particionados en 4 grupos de 32 SPs, manejados cada uno por un planificador de *warps*. Los SMs están agrupados en 4 GPC (*Graphics Processing Cluster*) y cada GPC contiene 4 SMs. Cada SM a su vez tiene incorporadas 8 unidades LD/ST y 8 unidades SFUs. 14

1.8. El diseño de un SM en Volta GV100. La arquitectura Volta GV100 está compuesta por 6 GPCs y cada uno de ellos contiene 14 SMs. Un SM en GV100 es particionado en cuatro bloques de procesamiento donde cada uno tiene 8 unidades LD/ST, 1 unidad SFU, un planificador de *warps* y 2 *Tensor Cores* para el procesamiento de matrices en aplicaciones de *machine learning* 15

1.9. La arquitectura GCN tiene 8 motores de computación asíncronos (ACEs) para manejar hasta 8 colas de cómputo y operar paralelamente con un procesador gráfico y los motores DMA. 17

1.10. Microarquitectura MIC (many-integrated-core) para los modelos. Una red de anillo bidireccional de alto rendimiento conecta núcleos de procesamiento, controladores de memoria y un módulo para la lógica de la conexión con el bus PCIe. 18

1.11. Microarquitectura MIC (many-integrated-core) Knights Landing KNL. KNL contiene 36 bloques o *tiles* interconectados mediante una malla 2D de coherencia de caché. Cada *tile* contiene 2 núcleos. Además proporciona controladores de memoria DRAM multicanal (MCDRAM) y memoria DDR (DDRMC) y soporte para bus PCIe 3.0. 20

1.12. Una GPU está constituida por un conjunto de multiprocesadores (SMs). Un programa CUDA es particionado en bloques de hilos que se ejecutan de forma independiente, por lo que idealmente una GPU con más multiprocesadores puede ejecutar automáticamente el mismo programa CUDA en menos tiempo que una GPU con menos multiprocesadores. 22

1.13. Jerarquía de Memoria en CUDA. 23

1.14. Modelo de plataforma OpenCL con un host y uno o más dispostivos OpenCL. Cada dispositivo OpenCL tiene uno o más unidades de cómputo (CU) y cada una de ellas uno o más elementos de procesamiento (PE). 33

1.15. Ejemplo de espacio 2D de N instancias del *kernel*. Las instancias se reparten en 4x4 *workgroups* (recuadro rojo) y cada *workgroup* está compuesto por 5x5 instancias del kernel (recuadro verde). . . 34

1.16. Resumen del modelo de memoria en OpenCL y como las distintas regiones de memoria iteractúan entre si. 39

2.1. Esquema de lanzamiento para CUDA y aceleradores con dos motores DMA. Este esquema utiliza un stream para cada tarea. El hilo del *host* lanza los comandos agrupándolos por tareas. Las dependencias internas de una tarea son mantenidas por medio de la sincronización implícita impuesta por el stream en los comandos que son lanzados dentro de él. 56

2.2. Esquemas de lanzamiento en OpenCL para aceleradores con un motor DMA y sin sincronización implícita. El número de colas empleadas en estos esquemas puede variar dependiendo de si se habilita CKE o no. En ambos esquemas, el hilo del *host* lanza los comandos agrupándolos por tipo. Además, las dependencias internas de una tarea tienen que ser manejadas por el hilo del *host* a través de eventos OpenCL. 57

2.3. Esquemas de lanzamiento para OpenCL y aceleradores con dos motores DMA y sin sincronización implícita. El número de colas empleadas en estos esquemas puede variar dependiendo de si se habilita CKE. OpenCL asocia las colas pares e impares a motores DMA diferentes, por lo tanto los comandos *HtD* y *DtH* se lanzan en las CQs 0 y 1 respectivamente. El hilo del *host* lanza los comandos agrupándolos por tareas. Además, este se encarga también de mantener las dependencias internas de una tarea mediante eventos OpenCL asociados a los comandos. 58

2.4. Rendimiento alcanzado por las transferencias de memoria cuando se usa memoria paginable y memoria no paginable. Las transferencias han sido realizadas para CUDA y OpenCL, desde 10 MB hasta 120 MB, sobre un bus PCIe 2.0 y para el acelerador NVIDIA K20c. 61

2.5. Ejemplo de simulación del solapamiento de transferencias. El modelo utiliza un nuevo parámetro λ que varía entre 0 y 1 para simular el grado de solapamiento entre las transferencias. 62

2.6. Error relativo en valor absoluto de la predicción para transferencias bidireccionales con distintos grados de solapamiento. La figura muestra los valores del error cometido para CUDA (NVIDIA K20c) y OpenCL (AMD R9). En la figura se consideran tres modelos: modelo sin solapamiento, modelo con solapamiento parcial y modelo con solapamiento completo. 64

2.7. Las flechas verdes entre comandos de diferentes colas, representan las dependencias internas de una tarea. Las flechas rojas simulan la dependencia cuando se envían comandos *HtD* y *DtH* en aceleradores con un motor DMA (los comandos *DtH* son enviados justo después de que todos los comandos *HtD* han sido enviados). Los comandos son insertados en sus respectivas colas y lanzados en ese orden. Los comandos que están actualmente en simulación son marcados con rectángulos azules con líneas discontinuas. Por el contrario, los comandos que han sido simulados son representados mediante cajas de color blanco. 66

2.8. Ejemplos de la simulación para OpenCL y CUDA, de la ejecución de varios comandos pertenecientes a tres tareas diferentes. Las líneas discontinuas rojas verticales identifican los pasos de la simulación. La información calculada en estos tres pasos de simulación se muestra en las tablas de abajo, que son válidas para entornos OpenCL y CUDA. En cada paso de la simulación, los comandos listos en la cabeza de las colas FIFO son identificados y se calculan sus tiempos de ejecución de inicio y fin. 67

2.9. Media geométrica del error de predicción para CUDA en los aceleradores NVIDIA K20c (a) y GTX 980 (b), para todas las permutaciones de tareas sintéticas en cada benchmark. Se presentan dos valores: *HyperQ* y *Single Queue*. Hyper-Q es una característica exclusiva de los aceleradores de Nvidia y emplea 32 colas hardware disponibles en el acelerador. La configuración de *Single Queue* solo emplea 1 cola, evita el reordenamiento y deshabilita las transferencias concurrentes en la misma dirección. 77

2.10. Media geométrica del error de predicción para CUDA en los aceleradores NVIDIA K20c (a) y GTX 980 (b), para todas las permutaciones de tareas reales en cada benchmark. Se presentan dos valores: *HyperQ* y *Single Queue*. Hyper-Q es una característica exclusiva de los aceleradores de Nvidia y emplea 32 colas hardware disponibles en el acelerador. La configuración de *Single Queue* solo emplea 1 cola, evita el reordenamiento y deshabilita las transferencias concurrentes en la misma dirección. 78

2.11. Media geométrica del error de predicción para OpenCL en los aceleradores AMD R9 (a), Intel Xeon Phi (b) y NVIDIA K20c (c), para todas las permutaciones de tareas sintéticas en cada benchmark. 79

2.12. Media geométrica del error de predicción para OpenCL en los aceleradores AMD R9 (a), Intel Xeon Phi (b) y NVIDIA K20c (c), para todas las permutaciones de tareas reales en cada benchmark. 80

3.1. Los hilos productores (*workers*) insertan la información de las tareas en un buffer compartido junto con el hilo proxy. Este buffer es consultado por el hilo proxy para recoger las tareas a ejecutar y realizar su planificación. 86

3.2. El hilo proxy detecta las tareas listas en el buffer compartido y construye un *TG* de tareas a ejecutar. Las tareas del *TG* son reordenadas y se lanzan sus respectivos comandos a las CQs correspondientes. Los comandos son enviados al acelerador por el hilo proxy empleando tres CQs (un acelerador con dos motores DMA). Los eventos de los últimos comandos *DtH* son recogidos en una lista compartida entre el hilo proxy y un hilo de sincronización. . . 87

3.3. El hilo proxy crea una lista de eventos insertados junto con los comandos *DtH*. Esta lista es compartida con un hilo de sincronización. Los eventos de esta lista indican cuando los comandos *DtH* han finalizado. Cuando el hilo de sincronización detecta un cambio de estado en algún evento de esta lista, este activa para su ejecución en el buffer compartido, la siguiente tarea del *worker* correspondiente. 88

3.4. Ejemplo de diagrama de clases para la definición de las tareas MatrixMult y VectorAddition. La clases de las tareas *MatrixMult* y *VectorAddition* son hijas de una clase abstracta llamada *Task*. La clase *Task* contiene métodos virtuales puros que definen la interfaz que tienen que implementar sus clases hijas. 89

3.5. Speedup conseguido con tareas sintéticas (arriba) y tareas reales (abajo), para cada benchmark y acelerador, con respecto a la peor permutación. Todos los experimentos han sido desarrollados usando la configuración de ejecución *Single Queue*. El speedup máximo es conseguido por la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio y el speedup de la heurística se calcula usando el orden obtenido por nuestra heurística. 99

3.6. Speedup conseguido con tareas sintéticas (arriba) y tareas reales (abajo), para cada benchmark y acelerador, con respecto a la peor permutación. Todos los experimentos han sido desarrollados usando la configuración de ejecución *Hyper-Q*. El speedup máximo es conseguido por la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio y el speedup de la heurística se calcula usando el orden obtenido por nuestra heurística, cuando se ejecuta con la configuración *Single Queue*. 100

3.7. Speedups conseguidos usando los tiempos de la Tabla 3.5, con respecto a la peor permutación usando la configuración *Hyper-Q* (izquierda) y la peor permutación usando la configuración *Single Queue* (derecha). 101

3.8. Resultados de concurrencia para benchmarks sintéticos (izquierda) y benchmarks reales (derecha). Cada columna representa el porcentaje de tiempo en que se están ejecutando concurrentemente uno, dos y tres comandos, para las GPUs K20c y GTX 980, usando las configuraciones de *Hyper-Q* y *Single Queue*. 105

3.9. Speedups con CKE conseguidos en el acelerador AMD R9, con tareas sintéticas y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística. 108

3.10. Speedups con CKE conseguidos en el acelerador NVIDIA K20c, con tareas sintéticas y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística. 109

3.11. Speedups con CKE conseguidos en el acelerador Intel Xeon Phi, con tareas sintéticas y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística. 110

3.12. Speedups con CKE conseguidos en el acelerador AMD R9, con tareas reales y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística. 111

3.13. Speedups con CKE conseguidos en el acelerador NVIDIA K20c, con tareas reales y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística. 112

3.14. Speedups con CKE conseguidos en el acelerador Intel Xeon Phi, con tareas reales y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística. 113

3.15. Speedups sin CKE conseguidos con tareas reales, para cada benchmark en el acelerador AMD R9. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística. 114

3.16. Speedups sin CKE conseguidos con tareas reales, para cada benchmark en el acelerador NVIDIA K20c. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística. 115

3.17. Speedups sin CKE conseguidos con tareas reales, para cada benchmark en el acelerador Intel Xeon Phi. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística. 116

3.18. Media geométrica del speedup máximo, mínimo y medio para los experimentos de los benchmarks reales. 116

4.1. Representación del lanzamiento de tareas en un acelerador como un problema *Flow Shop* de 3 máquinas. Los comandos *HtD* se ejecutan en la máquina 0, los comandos *K* en la máquina 1 y los comandos *DtH* en la máquina 2. Los comandos del trabajo k preceden a los comandos del trabajo $(k+1)$ en todas las máquinas. $p_{i(j)}$ corresponde al tiempo de procesamiento del trabajo j en la máquina i. 118

4.2. Ejemplo del lanzamiento de 4 tareas independientes en una GPU NVIDIA K20c. Las tareas *Matrix Multiplication*, *Black Scholes*, *Matrix Transposition* y *Vector Addition* se lanzan por los streams 13, 14, 15 y 16 respectivamente. La imagen correspondiente a la permutación I muestra el orden seleccionado por una heurística de planificación, mientras que la imagen de la permutación II corresponde al orden seleccionado por otra heurística. La reducción del *makespan* se debe únicamente a los diferentes órdenes de lanzamiento de tareas. 124

4.3. Porcentaje de error cometido por el modelo comparado con la ejecución real de los en las GPUs K20c (arriba), GTX 980 (centro) y Titan X (abajo) 128

4.4. Diagramas *box plot* del *speed-up* sobre la mediana del *makespan* de 10626 benchmarks de 4 tareas en las GPUs K20c (arriba), GTX980 (centro) y Titan X (abajo). La marca central corresponde a los valores de la media, las cajas abarcan el segundo y tercer cuartil, los bigotes se extiende hasta cubrir el 99,3 % de los valores, los valores atípicos se muestran individualmente con signos '+' y se usan muescas en las marcas de la media para intervalos de comparación. Se muestran los resultados para cada una de las heurísticas (SI, NEH, SQ y NEH-GPU) y para el mejor *makespan* (BEST). 132

4.5. Diagramas *box plot* del *speed-up* sobre la mediana del *makespan* de 10626 benchmarks de 4 tareas en las GPUs K20c (arriba), GTX980 (centro) y Titan X (abajo). Se muestran los resultados para la heurística NEH-GPU y diferentes combinaciones de tareas DK y DT (la columna de cajas más a la izquierda solamente considera benchmarks con 4 tareas DT, la siguiente columna de cajas benchmarks con 1 tarea DK y 3 tareas DT y así en adelante. 133

4.6. Proximidad a los mejores valores con diferentes número de tareas. 134

4.7. Proximidad a los mejores valores con diferente proporción de tareas DK. 135

4.8. Proximidad a los mejores valores con diferentes arquitecturas GPU. 135

Índice de tablas

1.1. Resumen de las características hardware de las distintas arquitecturas NVIDIA. La característica de *Solapamiento de Comandos* se refiere a la capacidad de la GPU de solapar comandos de transferencia de memoria con comandos de computación. 8

1.2. Equivalencia entre las terminologías usadas en entornos CUDA y OpenCL para el modelo de ejecución. 34

1.3. Equivalencia entre las terminologías usadas en entornos CUDA y OpenCL para el modelo de memoria. 38

2.1. Número de motores de DMA para los aceleradores NVIDIA K20c, GTX 980 y Titan X, Intel Xeon Phi y AMD R9. Estos aceleradores son los utilizados en esta tesis. 55

2.2. Porcentajes máximo, mínimo y medio del error relativo de predicción para todos los grados de solapamiento de transferencias bidireccionales usando tres modelos: modelo sin solapamiento, modelo con solapamiento parcial y modelo con solapamiento completo. . . 64

2.3. Tareas sintéticas de kernel dominante usadas en nuestros benchmarks. Los comandos *HtD*, *K* y *DtH* se definen con una fración de tiempo con respecto a una unidad de tiempo. La unidad de tiempo es 10 ms. 71

2.4. Tareas sintéticas de transferencias dominantes usadas en nuestros benchmarks. Los comandos *HtD*, *K* y *DtH* se definen con una fración de tiempo con respecto a una unidad de tiempo. La unidad de tiempo es 10 ms. 71

2.5. Benchmarks sintéticos usados. Cada benchmark se define con una etiqueta BKX, donde X es el porcentaje de tareas de kernel dominante en cada benchmark. 72

2.6. Tareas usadas en los benchmarks reales. Las tareas han sido seleccionadas según su clasificación como de kernel dominante o de transferencias dominantes. Las tareas CONV, DCT y FWT pueden presentar diferente comportamiento según el acelerador y el entorno de programación utilizados. De esta manera, las tareas DCT y FWT pueden ser de transferencias dominantes o de kernel dominante, si son ejecutadas en los aceleradores AMD R9, NVIDIA K20c, o Intel Xeon Phi respectivamente. Similarmente, la tarea CONV puede tener diferente comportamiento dependiendo de sus parámetros de entrada. 74

2.7. Rango de tiempos de ejecución para los comandos *HtD*, *K* y *DtH* pertenecientes a las tareas de los benchmarks reales en OpenCL. El rango de los tiempos de ejecución para cada tarea se obtiene ejecutando la tarea correspondiente con diferentes tamaños de datos de entrada. 74

2.8. Rango de tiempos de ejecución para los comandos *HtD*, *K* y *DtH* pertenecientes a las tareas de los benchmarks reales en CUDA. El rango de los tiempos de ejecución para cada comando se obtienen ejecutando la tarea correspondiente con varios conjuntos de parámetros de entrada. Los kernels *CONV 1* y *CONV 2* corresponden al kernel *CONV* con un número de iteraciones mayor que 1 e igual que 1 respectivamente. 75

3.1. Definición de los métodos de la clase que representa a una tarea. . 88

3.2. Tareas usadas en los benchmarks reales para entornos CUDA. Estas tareas han sido seleccionadas según su clasificación de kernel dominante (DK) o de transferencias dominantes (DT). La tarea *CONV* se puede clasificar como de kernel dominante o transferencias dominantes dependiendo de sus parámetros de entrada. 95

3.3. Composición de tareas para cada benchmark en los experimentos de cuatro tareas, en entornos CUDA. 96

3.4. Tiempos de ejecución de los benchmarks compuestos por cuatro tareas reales, para las configuraciones *Single Queue* e *Hyper-Q*. Todas las permutaciones posibles en el orden de las tareas han sido ejecutadas para recoger el tiempo máximo, mínimo y medio de todas ellas. También se ha recogido el tiempo de ejecución para el orden de tareas predicho por nuestra heurística. 97

3.5. Tiempos de ejecución máximo, mínimo y medio usando conjuntos de datos mayores, para los benchmarks de tareas reales en la GPU K20c, para las configuraciones *Hyper-Q*, *Single Queue* y nuestra heurística. 101

3.6. Composición de tareas para cada benchmark empleando, 4, 8 y 16 tareas. 102

3.7. Resultados para los benchmarks con tareas sintéticas y reales, usando 4, 6, 8 y 16 CQs, en las GPUs K20c y GTX980. Los valores mostrados son la media geométrica de las aceleraciones de nuestra heurística ejecutada con la configuración *Single Queue*, con respecto a la ejecución con la configuración *Hyper-Q*. 103

3.8. Tiempo medio de planificación empleado por el hilo proxy ejecutándose en una CPU Intel Core(TM)2 Quad, para benchmarks sintéticos usando 4, 6, 8 y 16 tareas. Además se muestra el tiempo de ejecución medio en una GPU K20c de un grupo de 4, 6, 8 y 16 tareas. 103

3.9. Tareas usadas en los benchmarks reales para entornos OpenCL. Las tareas han sido seleccionadas según su clasificación de kernel dominante o de transferencias dominantes. Las tareas DCT y FWT pueden tener diferente comportamiento según el acelerador utilizado. De esta manera, ambas tareas pueden ser de transferencias dominantes o de kernel dominante, si son ejecutados en el acelerador AMD R9 y NIVIDA K20c, o Intel Xeon Phi respectivamente. . 106

3.10. Tiempo medio de planificación en CPU empleado por el hilo proxy ejecutándose en una CPU Intel Core 2 Quad para benchmarks con 4, 6 y 8 tareas sintéticas. El tiempo medio de ejecución en GPU también se muestra para un grupo de tareas de 4, 6 y 8 tareas sintéticas en la GPU NVIDIA K20c. 111

4.1. Notación de la Teoría de Planificación 120

4.2. Tiempo de ejecución en milisegundos de los comandos de las tareas *Matrix Multiplication* (MM), *Black-Scholes* (BS), *Matrix Transposition* (TM) y *Vector Addition* (VA) usadas en el ejemplo de la Figura 4.2. La columna *Total* muestra el *makespan* de la tarea cuando se ejecuta sola. 125

4.3. *Makespan* parciales obtenidos por las heurísticas NEH y NEH-GPU al lanzar las tareas *Matrix Multiplication* (MM), *Black-Scholes* (BS), *Matrix Transposition* (TM) y *Vector Addition* (VA) usadas en el ejemplo de la Figura 4.2 del ejemplo de la Figura 4.2. 125

4.4. Selección de parámetros de entrada, de las tareas de la Tabla 3.2, para la GPU K20c. La tabla muestra para cada tarea los tiempos de ejecución en milisegundos de los comandos de transferencia no solapados (HtD y DtH) y solapados (oHtD y oDtH), así como también, del tiempo de los comandos *kernel* (K). 128

4.5. Selección de parámetros de entrada, de las tareas de la Tabla 3.2, para la GPU GTX 980. La tabla muestra para cada tarea los tiempos de ejecución en milisegundos de los comandos de transferencia no solapados (HtD y DtH) y solapados (oHtD y oDtH), así como también, el tiempo de los comandos *kernel* (K). 129

4.6. Selección de parámetros de entrada, de las tareas de la Tabla 3.2, para la GPU Titan X. La tabla muestra para cada tarea los tiempos de ejecución en milisegundos de los comandos de transferencia no solapados (HtD y DtH) y solapados (oHtD y oDtH), así como también, el tiempo de los comandos *kernel* (K). 130

4.7. Speed-up de la mediana de cada una de las heurísticas (SI, NEH, SQ y NEH-GPU) y para el mejor *makespan* (BEST) en las tarjetas K20c, GTX 980 y Titan X. 131

1 Introducción

Las arquitecturas de computación de alto rendimiento (HPC) se han convertido en una herramienta clave para la investigación y desarrollo de aplicaciones en diversos campos científicos y técnicos. La incorporación de unidades de procesamiento especializadas, también llamadas aceleradores, conectadas mediante buses, como el bus PCIe, han incrementado el rendimiento conseguido por este tipo de arquitecturas. Basta comprobar la lista con las 500 supercomputadoras con mayor rendimiento del mundo [1] para ver que están dominadas por sistemas que incluyen estos aceleradores. Las plataformas constituidas mediante la combinación de CPUs y aceleradores se denominan heterogéneas ya que incluyen arquitecturas con distintos conjuntos de instrucciones. Estos sistemas intentan combinar las ventajas de rendimiento en términos de latencia de las CPU, con el rendimiento en términos de productividad de los aceleradores.

Tradicionalmente la heterogeneidad en el contexto de la computación hace referencia a sistemas con distintos procesadores, con diferentes arquitecturas de conjuntos de instrucciones (ISA), y que tienen que ser controlados de forma distinta, aunque en la actualidad son cada vez más habituales los sistemas de computación que usan diferentes clases de procesadores (normalmente CPUs y GPUs). Estos sistemas mejoran su rendimiento no solo por añadir más procesadores, sino también por añadir procesadores distintos que poseen conjuntos de instrucciones diferentes. Esta incorporación de distintos procesadores permite mejorar la flexibilidad y eficiencia del sistema para la computación de tareas particulares. Aunque la capacidad de procesamiento de propósito general de las GPUs (aparte de sus bien conocidas capacidades gráficas de renderizado en 3D) permite realizar de forma eficiente cálculos matemáticos intensivos con grandes conjuntos de datos, las CPUs modernas son también capaces de alcanzar un rendimiento computacional similar aprovechando sus capacidades de paralelismo a nivel de

instrucción y baja latencia. Es por ello que la gestión de estas plataformas heterogéneas requiere de sistemas de planificación que identifiquen cuáles tareas son más adecuadas para cada procesador en base a criterios como la productividad, el consumo energético o la equidad, y se encarguen de coordinar su ejecución.

Normalmente, las computaciones masivamente paralelas son más apropiadas para los aceleradores, mientras que las secuenciales o moderadamente paralelas son más apropiadas para ejecutar en la CPU. Por lo tanto, en una aplicación algunas partes son ejecutadas en la CPU o *host*, mientras que otras son delegadas al acelerador o *device*. Esto conlleva una necesidad de transferencias de datos sobre el bus PCIe, normalmente entre diferentes espacios de memoria, lo que provoca una importante penalización en el rendimiento. El impacto de este cuello de botella puede disminuirse mediante el solapamiento de las transferencias a través del bus, con la ejecución en el acelerador de las diferentes tareas (también llamadas *kernels*). Es decir, dadas varias tareas que se pueden ejecutar concurrentemente, los comandos de transferencia y de computación pertenecientes a las distintas tareas se solapan para ocultar ese cuello de botella.

Algunos entornos de programación como CUDA [59] y OpenCL [34] proporcionan herramientas para hacer posible el solapamiento entre comunicación y computación. Estas herramientas se denominan *streams* en CUDA y colas de comandos en OpenCL, y se basan en el uso de comunicaciones asíncronas sobre el bus y el manejo de colas hardware para la computación en el acelerador. El solapamiento de comandos incrementa el número de tareas ejecutadas por unidad de tiempo en el acelerador (es decir, su productividad), y también reduce el tiempo en que el acelerador está desocupado.

1.1. Motivación. Ejemplo

La ejecución concurrente de varias aplicaciones puede dar lugar al envío de varios *kernels* al acelerador. Estas aplicaciones concurrentes puede estar localizadas de forma local o remota. Por ejemplo, algunos clusters de aceleradores emplean entornos de trabajo como rCUDA o CUDA MPS [19, 57], que permiten compartir un acelerador entre diferentes aplicaciones remotas. Consecuentemente, pueden coexistir en un instante dado una gran cantidad de tareas listas para ser lanzadas en el acelerador.

El soporte hardware para la ejecución concurrente de *kernels* varía de un acelerador a otro. Por ejemplo, en el caso de NVIDIA se denomina Hyper-Q a un conjunto de hasta 32 conexiones simultáneas, gestionadas por hardware, que

permiten lanzar tareas independientes, mientras que en el caso de AMD se usan hasta 8 *Asynchronous Compute Engines* (ACEs) que pueden gestionar hasta 8 colas hardware cada uno de ellos. Todos estos mecanismos se encargan de planificar las distintas tareas pero ninguno de ellos tienen en cuenta las transferencias de datos necesarias para completar las tareas. En la Sección 1.2 se detallará con más detenimiento el soporte hardware y software de los aceleradores actuales.

El rendimiento que se puede conseguir por la ejecución concurrente de un grupo de tareas es muy dependiente del orden de lanzamiento de las mismas. Este orden de lanzamiento interviene directamente en la política de planificación de recursos, afectando al solapamiento final entre las tareas involucradas.

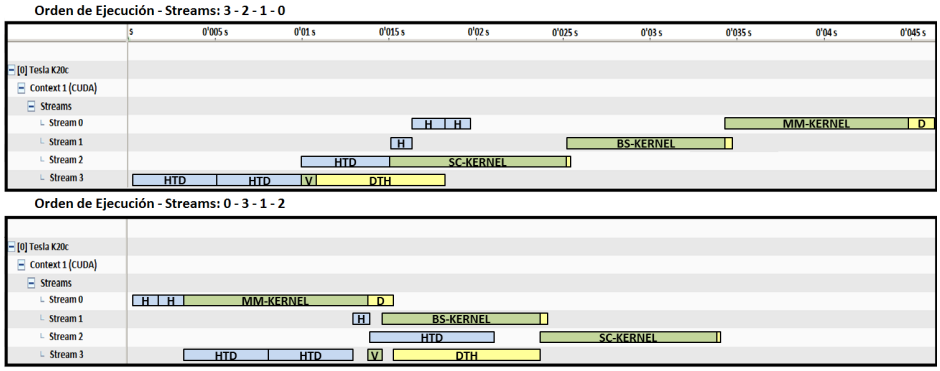


Figura 1.1: Dos ejemplos de ejecución concurrente de las mismas cuatro tareas, pertenecientes al SDK de CUDA, sobre un acelerador NVIDIA K20c. Las tareas Matrix Multiplication (MM-Kernel), Black Scholes (BS-Kernel), Separable Convolution (SC-Kernel) y Vector Addition (V) son lanzadas por los streams 0, 1, 2 y 3, respectivamente. Las transferencias *host to device* (HTD), los comandos de *kernels* y las transferencias *device to host* (DTH) son representadas mediante cajas azules, verdes y amarillas, respectivamente. Cada uno de los ejemplos muestran un orden de lanzamiento de tareas diferente.

Este hecho se puede apreciar en la Figura 1.1, donde se muestra un cronograma con el tiempo de ejecución de cuatro tareas empleando dos órdenes de lanzamiento distintos. Una tarea habitualmente esta compuesta por algunas (o ninguna) transferencias *host to device* (*HtD*), seguidas de uno o más comandos de ejecución, *kernels*, y por último algunas (o ninguna) transferencias *device to host* (*DtH*). Las transferencias *host to device* (*HtD*) son representadas en la Figura 1.1 mediante cajas de color azul. Estas transferencias envían datos que han sido previamente inicializados, desde la memoria de la CPU a la memoria del

acelerador. Una vez que todas las transferencias *HtD* de la tarea se han realizado, se pueden lanzar los comandos de ejecución o *kernels* en el acelerador. Los comandos *kernels* son representados en la Figura 1.1 mediante cajas de color verde. Estos comandos son los que verdaderamente realizan la computación en el acelerador, lanzando bloques de hilos donde todos los hilos realizan la misma computación pero con datos diferentes. Los aceleradores permiten la ejecución de varios comandos *kernels*, siempre que haya recursos hardware disponibles para la ejecución de todos ellos. El acelerador dispone de recursos hardware limitados como registros, cantidad de memoria, número máximo de hilos y de bloques de hilos soportado, etc., de forma que si alguno de ellos se consume al 100 % por parte de algún *kernel*, el acelerador no sería capaz de poder ejecutar otro *kernel* concurrentemente. Por último, una vez que todos los comandos *kernel* han finalizado su ejecución, los resultados de tal computación deben ser devueltos (si se requiere) a la memoria CPU. Esto se realiza mediante las transferencias *device to host* (*DtH*). Estas transferencias están representadas en la Figura 1.1 mediante cajas de color amarillo. Las transferencias *DtH* envían los resultados de la computación realizada en el acelerador, desde la memoria de este a la memoria de la CPU.

El envío de estas tareas a la GPU o el acelerador se realiza por medio de colas de comandos (o streams en terminología NVIDIA CUDA). Una cola de comandos es una secuencia de comandos que se ejecutan, normalmente en orden, en el acelerador. Por lo tanto, para cumplir las dependencias entre los comandos de una misma tarea, estos deberían ser lanzados a través de la misma cola de comandos. En el ejemplo mostrado en la Figura 1.1 se emplean cuatro colas de comandos (las nombramos como streams porque estamos en un entorno NVIDIA CUDA), a través de las cuales se lanza una tarea diferente por cada una de ellas. En el ejemplo se puede apreciar que los comandos de cada tarea se ejecutan en orden ya que pertenecen a la misma cola de comandos. Sin embargo, comandos pertenecientes a distintas cola de comandos sí pueden ejecutarse de forma concurrente y fuera de orden, ya que no existen dependencias entre ellos. De forma general, para conseguir la ejecución concurrente de tareas en un acelerador, estas deben lanzarse por distintas colas de comandos, y los comandos de las distintas tareas se ejecutarán de forma concurrente siempre que los recursos hardware lo permitan.

Las tareas de este ejemplo han sido seleccionadas del conjunto de benchmarks CUDA SDK [60] y lanzadas en un acelerador NVIDIA K20c. Más concretamente, las tareas son Matrix Multiplication (Stream 0), Black Scholes (Stream 1), Separable Convolution (Stream 2) y Vector Addition (Stream 3). Además, en este caso, las transferencias *HtD* y *DtH* también pueden solaparse entre si ya

que este acelerador tiene dos motores de copia o DMA (*Direct Memory Access*) que permiten transferencias opuestas simultaneas. Esta situación es habitual en cualquier acelerador actual y aumenta las posibilidades de ejecutar de forma concurrente varios comandos de tareas distintas en el acelerador. El ejemplo anterior muestra que el orden de lanzamiento de las tareas al acelerador puede tener un impacto importante en el tiempo de ejecución final, llegando a más de un 30 % de diferencia en este caso.

Suponiendo que una gran cantidad de tareas pueden estar listas para ser enviadas al acelerador, encontrar el orden de planificación óptimo que permita minimizar el tiempo total de ejecución requeriría comprobar todos los posibles órdenes. Para cada orden habría que hacer una ejecución previa, de todas las tareas o de al menos una parte de ellas, y obtener alguna estimación mediante un *profiler* o un simulador. Esta solución de búsqueda exhaustiva no es apropiada en tiempo de ejecución, ya que la comprobación de todos las posibles combinaciones para N tareas independientes conlleva evaluar $N!$ combinaciones posibles.

En esta tesis se aborda el problema de planificar un conjunto de tareas en un entorno HPC con aceleradores conectados mediante buses PCIe que son usados concurrentemente por diferentes aplicaciones. Para ello se desarrollan métodos que permitan, por un lado, estimar de forma precisa la ejecución de un grupo de tareas en un acelerador y, por otro lado, reducir el número de combinaciones a examinar para obtener una permutación, en tiempo de ejecución, lo más parecida posible a la óptima.

1.2. Entornos de Desarrollo

La aparición de aceleradores como las GPUs y las arquitecturas MIC (*Many Integrated Core*, p.ej., Intel Xeon Phi) dio lugar a la creación de entornos de programación que fueran capaces de explotar todas las funcionalidades ofrecidas por este tipo de procesadores. Existen varios entornos de programación para aceleradores, siendo NVIDIA-CUDA y OpenCL los más importantes y utilizados.

El incremento de la disponibilidad de transistores proporciona la posibilidad de integrar CPU y GPU en el mismo dispositivo, dando lugar a arquitecturas híbridas que pueden resolver las limitaciones de compartición de memoria entre ambos procesadores y haciendo su programación más fácil. Distintos fabricantes han lanzado sus arquitecturas híbridas al mercado como son AMD Kaveri [5], NVIDIA Tegra [55] e Intel con sus procesadores de séptima generación Kaby Lake [29]. Aunque las arquitecturas híbridas pueden resolver algunos problemas

presentes en las arquitecturas heterogéneas, también pueden presentar nuevos inconvenientes como por ejemplo un aumento de los retardos al acceder a memoria. En esta tesis nos hemos centrado en las arquitecturas heterogéneas y, en las siguientes secciones, daremos una visión general de estas arquitecturas, abordando aspectos como el bus de interconexión con los aceleradores, las arquitecturas de los aceleradores más habituales en la actualidad y los interfaces de programación más extendidos.

1.2.1. Puerto PCI Express (Peripheral Component Interconnect Express)

PCI Express, abreviado como PCIe o PCI-e, es un bus serie de expansión de alta velocidad, diseñado para reemplazar a los buses anteriores PCI, PCI-X y AGP. El bus PCIe proporciona varias mejoras comparado con los buses anteriores. Estas mejoras incluyen un mejor rendimiento, menor consumo de energía y un mecanismo para el informe y detección de errores en las transmisiones de datos.

Una de las diferencias claves entre el bus PCI Express y los buses anteriores es su topología. PCI utiliza una arquitectura compartida de bus paralelo, en la que el *host* y todos los dispositivos comparten un conjunto de direcciones, datos y líneas de control. Por el contrario, PCIe se basa una topología punto a punto, con enlaces serie que conectan individualmente cada dispositivo al *host*. Debido a su topología compartida, el bus PCI es arbitrario (en el caso de varios *hosts* o *masters*), y limitado a un único *host* o *master* en un tiempo dado, en una sola dirección. Además, el bus PCI tiene un sistema de frecuencia de reloj que se limita a la frecuencia de reloj del dispositivo más lento. Sin embargo, un bus PCIe soporta una comunicación *full-duplex* entre dos puntos, sin ninguna limitación en el acceso concurrente por parte de ambos.

En PCIe 1.1 cada carril o línea punto a punto transporta hasta 250 MB/s en cada dirección. PCIe 2.0 dobla esta tasa hasta 500 MB/s y PCIe 3.0 alcanza hasta 1 GB/s por cada carril (2 GB/s bidireccional). Por tanto, PCIe 3.0 logra en el caso de x16 un máximo teórico de 16 GB/s direccionales y 32 GB/s bidireccional.

PCIe está pensado para ser usado solo como un bus local, aunque existen extensores capaces de conectar múltiples placas base mediante cables de cobre o incluso fibra óptica. Este conector es usado sobre todo para conectar GPUs. Las últimas GPUs discretas de fabricantes como AMD y NVIDIA utilizan la versión 3.0 del bus PCIe para la interacción con el *host*. Esta interfaz entre el *host* y la GPU es un cuello de botella, particularmente para cargas de trabajo de propósito general, donde grandes cantidades de datos tienen que ser transferidas entre

los dos procesadores. Para este fin, las actuales GPUs tienen dos motores DMA bidireccionales, por lo que dos flujos de datos pueden usar concurrentemente ambas direcciones del bus PCIe y usar eficientemente el ancho de banda disponible. Además, incorporan una unidad de manejo de memoria de E/S, que puede de forma transparente mapear direcciones del *host* en la GPU. Esto significa que los motores DMA pueden acceder a memoria paginable del *host* sin penalización alguna.

1.2.2. Arquitecturas

Los aceleradores más habituales en la actualidad, en entornos de computación de alto rendimiento, son los fabricados por NVIDIA, AMD e Intel. En las próximas secciones describiremos la evolución y características más importantes de cada uno de ellos, centrándonos especialmente en los aceleradores de NVIDIA.

Aceleradores NVIDIA

NVIDIA ha sido en los últimos años uno de los principales referentes en el diseño de GPUs. Este fabricante lanzó en el año 2007 CUDA [59], que son las siglas de *Compute Unified Device Architecture*, haciendo accesibles los recursos de computación de la GPU a los programadores de software. De esta manera, las GPUs de NVIDIA se convirtieron en una importante alternativa para la computación de altas prestaciones. Las distintas arquitecturas lanzadas por el fabricante han ido evolucionando desde el año 2007 hasta nuestros días. Seis son las microarquitecturas lanzadas por el fabricante hasta el año 2017: Tesla, Fermi, Kepler, Maxwell, Pascal y Volta. La Tabla 1.1 muestra ordenadas, en orden cronológico de izquierda a derecha, las características hardware más relevantes de las diferentes arquitecturas.

En el año 2007, NVIDIA lanza su arquitectura GPU denominada Tesla [45]. Los SMs (*Streaming Multiprocessors*) son el corazón de las arquitecturas NVIDIA. Contienen tanto recursos de computación como de memoria. Las operaciones aritméticas son ejecutadas en un vector de SPs (*Streaming Processors*) o *CUDA cores*, mientras que otras instrucciones más complejas son ejecutadas en las SFUs (*Special Functions Units*). Los SMs también incorporan recursos de memoria compartida, de constantes y de cachés. El diseño de los SMs revela grandes diferencias a lo largo de las diferentes arquitecturas. La Figura 1.2 muestra el diseño de los SM para la arquitectura Tesla. Esta arquitectura está compuesta por un *Texture Processor Cluster* (TPC). Cada TPC contiene 2 SMs que están

Arquitectura	Tesla	Fermi	Kepler (GK110)	Maxwel (GM204)	Pascal (GP104)	Volta (GV100)
SMs	hasta 16	hasta 16	hasta 15	hasta 16	hasta 20	hasta 84
Núcleos/SM	8	32	192	hasta 128	hasta 128	hasta 128
Planif. de Warps/SM	1	2	4	4	4	4
Mem. Compartida/SM	16KB	48KB	64KB	96KB	96KB	128KB
Registros/SM	8192	32768	65536	65536	65536	65536
Hilos/Warp	32	32	32	32	32	32
Hilos/Bloque	512	1024	1024	1024	1024	1024
Bloques/SM	8	8	16	16	32	32
Kernels Concurrentes	No	Si	Si	Si	Si	Si
Solapamiento de Comandos	No,Si	Si	Si	Si	Si	Si

Tabla 1.1: Resumen de las características hardware de las distintas arquitecturas NVIDIA. La característica de *Solapamiento de Comandos* se refiere a la capacidad de la GPU de solapar comandos de transferencia de memoria con comandos de computación.

compuestos a su vez por 8 SPs y 2 SFUs. Para ejecutar eficientemente cientos de hilos en paralelo, el hardware de los SMs implementa una arquitectura SIMT (*Single-Instruction, Multiple-Thread*). Cada SM crea, maneja, planifica y ejecuta grupos de 32 hilos conocidos como *warps*. Los hilos del *warp* son mapeados por el SM a los SPs. El planificador de *warps* lanza instrucciones de un *warp* en los SPs cada 4 ciclos de reloj. Los SPs son los encargados de la ejecución de los hilos, mientras que las SFUs soportan la computación de funciones complejas (senos, cosenos, raíces).

En el año 2010, NVIDIA da un paso adelante en el diseño de GPUs y lanza una nueva arquitectura denominada Fermi [52]. Fermi introduce varias innovaciones en la arquitectura que hace a los SMs más eficientes y más fáciles de programar. La Figura 1.3 muestra el diseño de un SM en Fermi. Esta arquitectura está compuesta por 16 SMs y cada SM tiene 32 SPs (cuatro veces más que Tesla). Por lo tanto, Fermi tiene un número total de SPs igual a 512. Cada SP ejecuta una instrucción entera y de punto flotante por ciclo de reloj y por hilo. Cada SM tiene 16 unidades LD/ST (unidades de carga y almacenamiento), 4 SFUs y un bloque de 64 KB de memoria compartida entre todos los SPs del SM. Al igual que en Tesla, los *warps* son planificados por cada SM. Fermi presenta un doble planificador de *warps* por SM, que permite la ejecución concurrente de dos *warps*. El doble planificador de *warps* selecciona 2 warps y lanza una instrucción de cada uno de ellos a un grupo de 16 SPs. Fermi no necesita comprobar las dependencias dentro del cauce de instrucciones, ya que los *warps* se ejecutan de forma independiente.

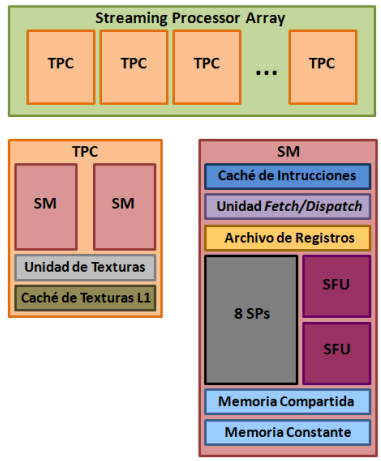


Figura 1.2: Arquitectura Tesla. Esta arquitectura está compuesta por un *Texture Processor Cluster* (TPC). Cada TPC contiene 2 SMs (*Streaming Multiprocessors*) que están compuestos a su vez por 8 SPs (*Streaming Processors*) y 2 SFUs (*Special Functions Units*).

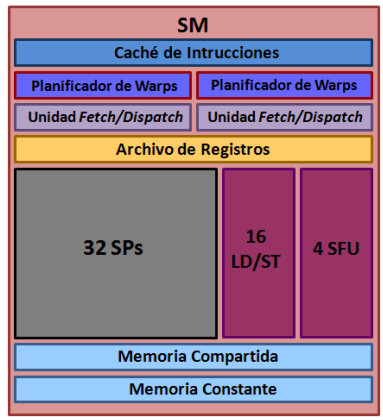


Figura 1.3: Diseño de un SM (*Streaming Multiprocessor*) en la arquitectura Fermi. Esta arquitectura está compuesta por 16 SMs que están compuestos a su vez por 32 SPs (*Streaming Processors*), 16 unidades LD/ST (Load and Store Units) y 4 SFUs (*Special Functions Units*). Fermi presenta un doble planificador de *warps* para ejecutar concurrentemente dos *warps* en el mismo SM.

Una de las características más importantes de la arquitectura Fermi es su planificador de hilos de dos niveles denominado *GigaThread Scheduler*. A nivel de chip, los motores de distribución de trabajo planifican bloques de hilos a los distintos SMs, mientras que a nivel de SM, cada planificador de *warp* distribuye conjuntos de 32 hilos a sus SPs. Gracias a este planificador de dos niveles, la arquitectura Fermi es capaz de soportar la ejecución concurrente de diferentes comandos de computación en la misma GPU.

La siguiente arquitectura tras Fermi fue lanzada en el año 2012 con el nombre de Kepler GK110 [53]. Esta arquitectura mejora a la anterior añadiéndole nuevas características hardware. Algunas de las características más importantes añadidas por Kepler son: una nueva arquitectura SM denominada SMX (*Next Generation Streaming Multiprocessor*), una solución que permite que varios hilos CPU compartan la misma GPU (Hyper-Q) y una nueva unidad de planificación de bloques de hilos denominada GMU (Grid Management Unit).

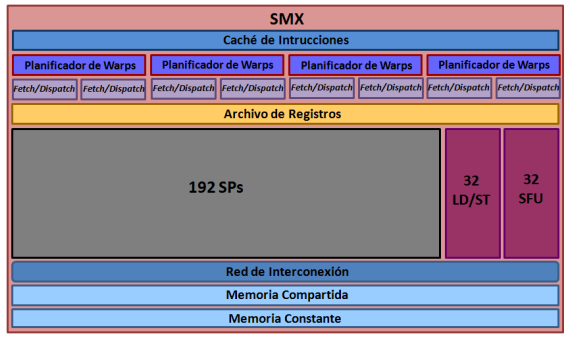


Figura 1.4: El diseño de un SM es mejorado en Kepler dando lugar al diseño denominado SMX (*Next Generation Streaming Multiprocessor*). Esta arquitectura está compuesta por 15 SMs que están compuestos a su vez por 192 SPs (*Streaming Processors*), 32 unidades LD/ST (Load and Store Units) y 32 SFUs (*Special Functions Units*). Kepler es capaz de ejecutar 4 *warps* en un SMX, debido a su cuádruple planificador de *warps*.

Cada una de las unidades SMX en la arquitectura Kepler y, concretamente, en su modelo GK110 contiene 192 SPs. El diseño SMX en Kepler GK110 es mostrado en la Figura 1.4. El objetivo del diseño de las unidades SMX fue, por un lado, reducir su consumo energético y, por el otro, incrementar significativamente el rendimiento en operaciones de doble precisión, ya que la aritmética de doble precisión es el corazón en muchas aplicaciones de computación de altas prestaciones. Otra característica novedosa en el diseño SMX es que cada SM tiene incorporado

un cuádruple planificador de *warps*, lo que permite lanzar y ejecutar concurrentemente hasta cuatro *warps*. Además, incorpora 32 unidades LD/ST y 32 SFUs. El tamaño de memoria compartida entre los SPs también se incrementa a 64KB.

Unos de los objetivos principales para obtener un buen rendimiento en GPU es realizar una planificación eficiente de distintas cargas de trabajo pertenecientes a distintos streams. Un stream es una secuencia de comandos que se ejecutan en orden en la GPU. Sin embargo, comandos localizados en diferentes streams pueden ejecutarse concurrentemente y fuera de orden. Fermi podía lanzar concurrentemente hasta 16 comandos de computación o *kernels* desde diferentes streams, pero todos esos streams eran después multiplexados sobre la misma cola de trabajo. Esto originaba una falsa dependencia en el interior del stream disminuyendo las capacidades de concurrencia de la GPU. La arquitectura Kepler en todos sus modelos mejora esta funcionalidad mediante una nueva característica denominada Hyper-Q. Hyper-Q incrementa el número total de conexiones (colas de trabajo) entre el *host* y la GPU, permitiendo hasta 32 conexiones simultáneas. Cada stream es manejado dentro de su propia cola hardware o de trabajo, las dependencias entre streams son optimizadas y operaciones dentro de un stream no bloquearán la ejecución de otros streams. La Figura 1.5 muestra un ejemplo de la falsa dependencia introducida en Fermi. En esta figura se pueden apreciar como Fermi y Kepler trabajan con tres streams. Cada ejemplo lanza nueve kernels repartidos en tres streams de la siguiente manera: A, B y C (stream 1), P, Q y R (stream 2) y X, Y y Z (stream 3). En el modelo de Fermi, los tres streams son multiplexados en una sola cola hardware, lo que origina que solamente las parejas de kernels (C, P) y (R, X) puedan ejecutarse concurrentemente. Sin embargo, el modelo de Kepler, al utilizar Hyper-Q, asocia cada stream a una cola hardware diferente, haciendo posible que todos los streams puedan ejecutarse concurrentemente.

Otras nuevas características en Kepler, como la capacidad de que la propia GPU genere más trabajo sobre si misma (denominado paralelismo dinámico), consigue incrementar la potencia de computación de Kepler sobre Fermi. En Fermi, existe solo un flujo unidireccional de trabajo desde el *host* a los SMs de la GPU mediante la unidad CWD (CUDA Work Distributor). Esto hace que cuando el *host* lanza un *grid* (conjunto) de bloques de hilos a la GPU, otro *grid* no podrá ejecutarse hasta que el anterior no haya terminado su ejecución. Kepler ha sido diseñada para mejorar el flujo de trabajo CPU-GPU, permitiendo a la GPU manejar eficientemente cargas de trabajo generadas tanto por el *host* como por ella misma. Para manejar ambas cargas de trabajo, se introduce una nueva unidad denominada GMU (Grid Management Unit). Esta unidad maneja y asigna prioridades a cargas de trabajo que son pasadas a la unidad CWD para después

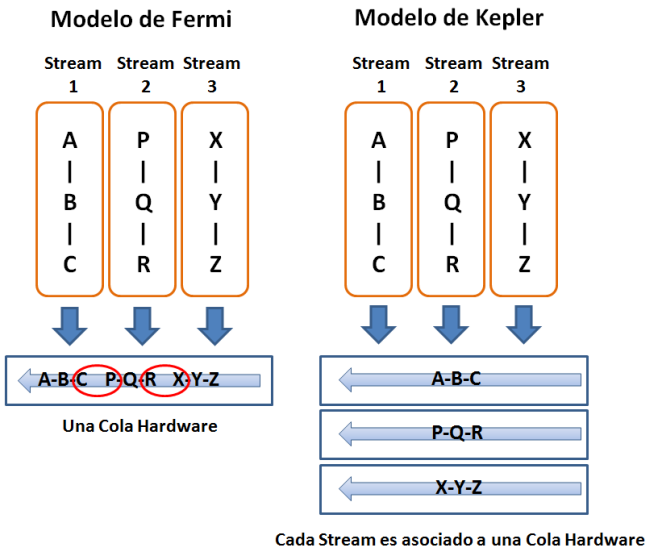


Figura 1.5: En el modelo de Fermi mostrado a la izquierda, solamente las parejas de kernels (C, P) y (R, X) pueden ejecutarse concurrentemente debido a la dependencia causada por el uso de una sola cola hardware. El modelo de Kepler utiliza Hyper-Q, que permite a todos los streams ejecutarse concurrentemente usando colas hardware separadas.

ser enviadas a los SMXs. La Figura 1.6 muestra a la izquierda y a la derecha el flujo de trabajo entre el *host* y la GPU para Fermi y Kepler respectivamente. La unidad CWD en Kepler mantiene *grids* que están listos para ser ejecutados, y es capaz de lanzar hasta 32 *grids*, que es el doble de capacidad que en Fermi. Un *grid* es un conjunto de bloques de hilos. La unidad CWD en Kepler se comunica con la unidad GMU mediante un enlace bidireccional, que permite a la GMU pausar el lanzamiento de nuevos *grids*, mantenerlos pendientes o suspenderlos hasta que se necesiten. Esta unidad también tiene una conexión directa con las unidades SMXs para permitir la creación de trabajo adicional por parte de *grids* que se están ejecutando.

NVIDIA siguió perfeccionando el diseño de sus arquitecturas GPUs, de manera que en el año 2014 da a conocer la arquitectura sucesora de Kepler denominada Maxwell [56]. Maxwell introdujo pocas características nuevas con respecto a Kepler, ya que se centró principalmente en la eficiencia energética. La caché L2 fue incrementada de 256 KB en Kepler a 2 MB en Maxwell, reduciendo así la necesi-

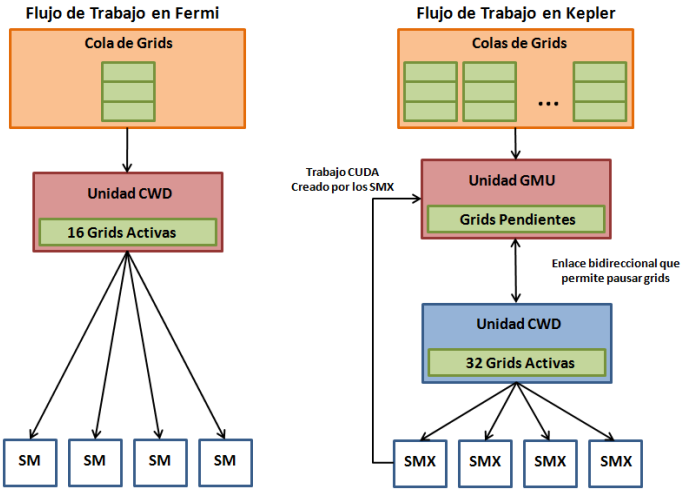


Figura 1.6: Flujo de trabajo entre el *host* y la GPU para Fermi (izquierda) y para Kepler (derecha). El rediseño del flujo de trabajo en Kepler muestra una nueva unidad llamada GMU (Grid Management Unit), que permite lanzar, pausar y supender las *grids* activas.

dad de ancho de banda de memoria. El bus de memoria fue reducido a 128 bits, para reducir el gasto de energía. Los SMs de Kepler fueron reestructurados y particionados adoptando el nombre de SMM (*Streaming Maxwell Multiprocessor*). Al igual que Fermi y Kepler, Maxwell está compuesta por un vector de clusters de procesamiento gráfico GPC (*Graphics Processing Clusters*). Concretamente la arquitectura Maxwell GM204 está compuesta por 4 GPCs que a su vez contienen 4 SMMs, lo que hace un total de 16 SMMs. En la Figura 1.7 se muestra el diseño de un SMM en la arquitectura Maxwell GM204. Un SMM contiene 4 planificadores de *warps*, y cada uno de ellos es capaz de enviar 2 instrucciones por *warp* cada ciclo de reloj. Mientras que en Kepler los 4 planificadores de *warps* mapeaban los *warps* en 192 SPs, en Maxwell cada planificador de *warps* se encarga solo de 32 SPs (128 *CUDA cores* por SMM). Además, cada partición de 32 SPs tiene un buffer de instrucciones, 8 unidades LD/ST y 8 unidades SFU. Aunque se reduce el número de núcleos, la mejora en la planificación proporcionada por el particionamiento de estos junto con el ahorro energético, hacen que el diseño de los SMM sea más eficiente.

Maxwell también adopta de Kepler la característica hardware de Hyper-Q, en la que se usan varias colas de trabajo para la comunicación con la GPU. La

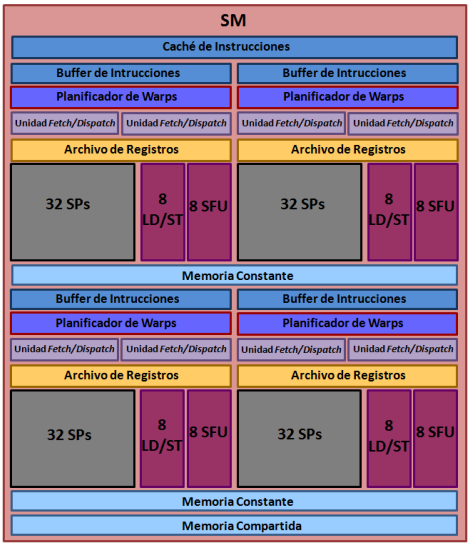


Figura 1.7: El diseño de un SM en Maxwell GM204 (diseño adoptado también por la arquitectura Pascal). El número de SPs es reducido de 192 en Kepler a 128 en Maxwell. El número de SPs son particionados en 4 grupos de 32 SPs, manejados cada uno por un planificador de *warps*. Los SMs están agrupados en 4 GPC (*Graphics Processing Cluster*) y cada GPC contiene 4 SMs. Cada SM a su vez tiene incorporadas 8 unidades LD/ST y 8 unidades SFUs.

unidad GMU de Kepler también es añadida a Maxwell para hacer posible la creación de trabajo por parte de la propia GPU.

El continuo avance de la computación de altas prestaciones y la programabilidad de las GPUs conduce a importantes mejoras tanto en los gráficos 3D como en la computación en GPU. Esta continua evolución hace que en el año 2016 NVIDIA lance una nueva arquitectura sucesora a Maxwell. Esta nueva arquitectura recibe el nombre de Pascal [54]. Pascal está compuesta por una configuración de GPC distinta a Maxwell. Concretamente, la arquitectura Pascal GP104 está compuesta por 4 GPCs que a su vez contienen 5 SMs, lo que hace un total de 20 SMs. La arquitectura Pascal hereda el diseño de SM de Maxwell (Figura 1.7), donde 32 SPs son repartidos entre 4 planificadores de *warps*. Con un total de 20 SMs, la arquitectura Pascal GP104 tiene un total de 2560 SPs (*CUDA cores*). Al igual que Maxwell, Pascal también tiene incorporadas las características de Hyper-Q y la unidad GMU.

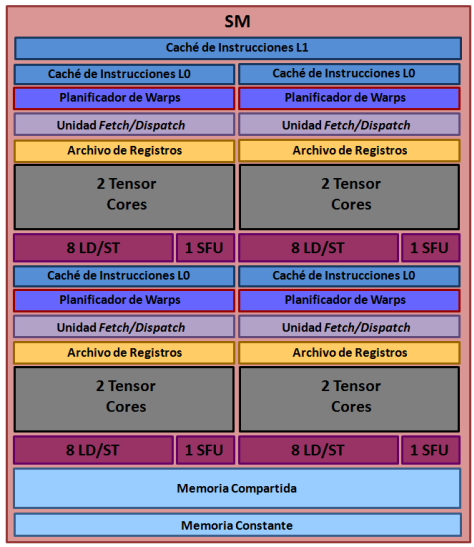


Figura 1.8: El diseño de un SM en Volta GV100. La arquitectura Volta GV100 está compuesta por 6 GPCs y cada uno de ellos contiene 14 SMs. Un SM en GV100 es particionado en cuatro bloques de procesamiento donde cada uno tiene 8 unidades LD/ST, 1 unidad SFU, un planificador de *warps* y 2 *Tensor Cores* para el procesamiento de matrices en aplicaciones de *machine learning*

Desde la primera arquitectura CUDA, en cada nueva generación de GPUs NVIDIA ha aportado un mayor rendimiento, un mejor consumo energético y añadido nuevas características hardware. En el presente, las GPUs NVIDIA están presentes en multitud de aplicaciones de alto rendimiento, centros de datos y, recientemente, en sistemas de aprendizaje automático (*machine learning*). Los sistemas de *machine learning* abarcan multitud de aplicaciones y campos científicos incluyendo plataformas de vehículos autónomos, reconocimiento del habla y de texto, diagnóstico de enfermedades, etc. Para cubrir la demanda de estos campos en constante crecimiento, NVIDIA da a conocer en Mayo de 2017 su nueva arquitectura denominada Volta [58]. La nueva arquitectura Volta incorpora nuevas innovaciones hardware destinadas a incrementar el rendimiento de aplicaciones de *machine learning*, además de proporcionar un alto rendimiento. Concretamente la arquitectura Volta GV100 está compuesta por 6 GPCs y cada GPC contiene 14 SMs, lo que hace un total de 84 SMs. En la Figura 1.8 se muestra el diseño de un SM en la arquitectura Volta GV100. Un SM en GV100 es particionado en cuatro bloques de procesamiento donde cada uno tiene 8 unidades LD/ST, 1

unidad SFU, un planificador de *warps* y 2 *Tensor Cores* para el procesamiento de matrices en aplicaciones de *machine learning*. Aunque un SM en Volta GV100 tiene el mismo número de registros que Pascal GP104, GV100 tiene más SMs, y de esta manera más registros en total. Por tanto, GV100 soporta más hilos, *warps* y bloques de hilos que GP104. Por último, Volta GV100 también incorpora las características de Hyper-Q y la unidad de GMU.

Aceleradores AMD

AMD, al igual que NVIDIA, ha tenido un diseño de arquitectura de GPUs que ha sido un referente en el diseño de aceleradores. Esta arquitectura es la llamada Graphics Core Next (GCN) [6]. La arquitectura GCN fue lanzada en el año 2011 como sucesora de la arquitectura Terascale y ha ido evolucionando cinco generaciones hasta la actualidad, donde se espera una próxima evolución (la sexta) para el año 2018. La arquitectura GCN se caracteriza por disponer de un procesador de comandos gráficos (*Graphics Command Processor*, GCP) que se encarga del renderizado, y unos núcleos de computación asíncrona (*Asynchronous Compute Engine*, ACE) que se encargan de enviar trabajo a las unidades de computación (*Compute Units*, CU) a través de hasta 8 colas independientes.

Las CUs son el bloque computacional básico de la arquitectura GCN. Cada CU incluye 4 unidades SIMD (Single Instruction Multiple Data) para procesamiento vectorial. Cada una de las unidades SIMD ejecutan 16 *workitems* con la misma instrucción, pero cada uno de ellos puede pertenecer a distintos *wavefronts*. Un *workitem* es una instancia del kernel que se ejecutará en el acelerador y un *wavefront* es un conjunto de 64 *workitems* . Las unidades SIMD tienen su propio contador de programa y son capaces de ejecutar hasta 10 *wavefronts*. En cada ciclo de reloj, 10 instrucciones de diferente tipo y pertenecientes a distintos *wavefronts* son descodificadas y lanzadas. De esta manera, se evita la sobrescritura en el cauce de ejecución de cada tipo de instrucción y se mantiene el orden de ejecución. Por lo tanto, una CU es capaz de manejar al mismo tiempo 40 *wavefronts* de distintos *workgroups* o *kernels*. Un *workgroup* es un conjunto de varios *wavefronts*.

Las GPUs discretas basadas en GCN (por ejemplo: AMD Radeon HD7870 y AMD R9 290) utilizan la versión 3.0 del bus PCI Express para la interacción con el *host*. Una ranura PCI Express 3.0 tiene 1 GB/s direccional y 2 GB/s bidireccional, por lo que logran en el caso de x16 un máximo teórico de 16 GB/s direccionales y 32 GB/s bidireccional. Esta interfaz entre el *host* y la GPU es un cuello de botella, particularmente para cargas de trabajo de propósito general, donde grandes cantidades de datos tienen que ser transferidas entre los dos

procesadores. Para este fin, GCN proporciona dos motores DMA bidireccionales, por lo que dos flujos de datos pueden concurrentemente usar ambas direcciones del bus PCIe y usar eficientemente el ancho de banda disponible. GCN incorpora una unidad de manejo de memoria de E/S, que puede transparentemente mapear direcciones del *host* en la GPU. Esto significa que los motores DMA pueden acceder a memoria paginable del *host* sin penalización alguna.

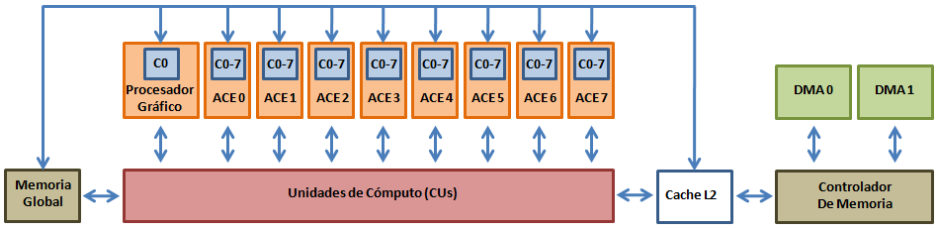


Figura 1.9: La arquitectura GCN tiene 8 motores de computación asíncronos (ACEs) para manejar hasta 8 colas de cómputo y operar paralelamente con un procesador gráfico y los motores DMA.

Diferentes flujos de comandos pueden ser manejados por GCN por medio de 8 núcleos de computación asíncronos (ACEs). La Figura 1.9 muestra como cada ACE puede analizar comandos entrantes de 8 colas de cómputo independientes, y enviar trabajo a las distintas unidades de procesamiento de la GPU. Cada cola de cómputo envía *wavefronts* sin esperar a que otra tarea haya finalizado, permitiendo que flujos independientes de comandos sean intercalados y ejecutados concurrentemente. Mientras que los ACEs son responsables de manejar los comandos de computación, el procesador de gráficos maneja los comandos destinados a gráficos y ambos pueden trabajar de forma paralela junto con los motores DMA. Cada ACE puede capturar comandos de la caché o de la memoria y formar colas de tareas con prioridad, las cuales están preparadas para su planificación. La tarea con mayor prioridad será enviada por el ACE para su ejecución si hay recursos hardware disponibles.

La arquitectura GCN puede tener varias tareas listas para planificar o ejecutándose concurrentemente. El número máximo de tareas que la arquitectura puede soportar en un instante de tiempo viene impuesto por los recursos hardware disponibles. Cuando una tarea es enviada para su ejecución, esta es descompuesta en un número de *workgroups* que son lanzados en CUs individuales para su ejecución. Cada ciclo, un ACE puede crear un *workgroup* y lanzar un *wavefront* del *workgroup* a las CUs.

Aunque los ACEs normalmente operan de forma independiente, pueden sin-

cronizarse y comunicarse por medio de la memoria caché o datos compartidos en memoria global. Esto significa que un ACE puede formar un grafo de tareas, donde tareas individuales pueden tener dependencias entre si. Por lo tanto, una tarea localizada en un ACE puede tener dependencia con otra tarea localizada en otro ACE diferente. Los ACEs pueden operar con distintas colas de tareas, parando una tarea y seleccionando la tarea siguiente de una cola diferente. Por ejemplo, si la tarea que actualmente está ejecutándose tiene que esperar debido a una dependencia, el ACE podría cambiar a otra cola de tareas que tiene tareas listas para su planificación. Esto se lleva a cabo descartando los *workgroups* de la antigua tarea y lanzando los *workgroups* de la nueva tarea en las unidades de cómputo.

Aceleradores Intel

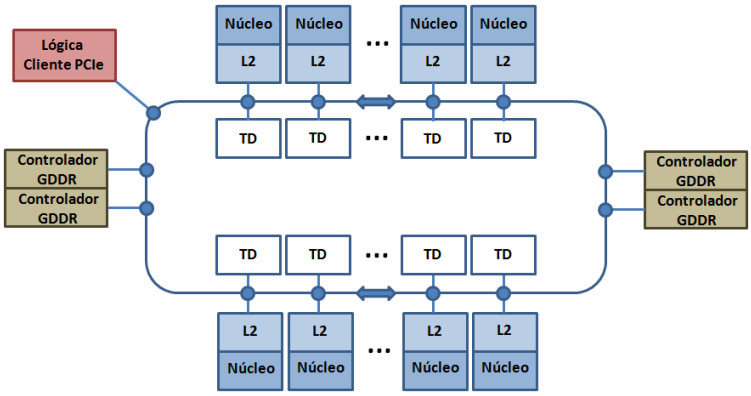


Figura 1.10: Microarquitectura MIC (many-integrated-core) para los modelos. Una red de anillo bidireccional de alto rendimiento conecta núcleos de procesamiento, controladores de memoria y un módulo para la lógica de la conexión con el bus PCIe.

Los aceleradores de Intel están basados en una arquitectura denominada *many-integrated-core* (MIC) [30], donde varios núcleos Intel Pentium residen en el mismo chip. Esta arquitectura sigue la filosofía de un diseño previo de Intel denominado Larrabee [67]. Posteriormente surgió la arquitetura MIC con su primer prototipo denominado Knights Ferry (KNF) [83]. KNF contiene 32 núcleos x86 de 1,2 GHz que soportan 4 hilos hardware cada uno y un pequeño cauce de ejecución en orden. Cada núcleo tiene 256KB de caché compartida L2, 32KB de caché de datos L1 y 32Kb de caché de instrucciones. Tanto los núcleos como

los controladores de memoria están interconectados entre sí mediante una red de anillo bidireccional. Todas las cachés son coherentes entre si junto con la memoria principal. Además, los núcleos de KNF comparten 1 o 2 GB de memoria principal GDDR5. La comunicación entre el *host* y KNF se realiza mediante un bus PCIe 2.0 y un solo motor DMA que soporta el movimiento asíncrono de datos entre el acelerador y el *host*.

La segunda evolución de la arquitectura MIC es denominada Knights Corner (KNC) [17] y es lanzada en el año 2013. KNC contiene 60 núcleos Intel Pentium de 1,09 GHz (61 en algunas versiones) y 8 GB de memoria principal GDDR5. Además, está provista de 32 KB de caché L1 destinada para datos e instrucciones y 512 KB de caché de datos L2 (compartida por todos los núcleos). En KNC, al igual que en KNF, los núcleos están interconectados por una red de anillo bidireccional y la comunicación entre el dispositivo y el *host* se realiza mediante un bus PCIe 2.0 y un solo motor DMA.

La Figura 1.10 muestra un esquema de la arquitectura MIC para KNF y KNC compuesta por núcleos de procesamiento, controladores de memoria, un módulo para la lógica de conexión con el PCIe y una red de anillo bidireccional de alto rendimiento para la interconexión de todos los módulos. Cada núcleo tiene una memoria caché L2 privada, que mantiene coherencia total mediante un directorio global distribuido (TD). Los controladores de memoria y la lógica de conexión con el PCIe proporcionan una interfaz de acceso directo con la memoria GDDR5 y el bus PCIe, respectivamente. Cada núcleo en MIC utiliza un pequeño cauce de ejecución en orden que ejecuta 4 hilos hardware, lo que significa que cada núcleo puede ejecutar concurrentemente instrucciones de 4 hilos o procesos. Esto ayuda a reducir las latencias producidas por los accesos de memoria.

La tercera iteración de la arquitectura MIC es conocida como Knights Landing (KNL) [70]. La Figura 1.11 muestra un esquema de la arquitectura KNL. KNL contiene 36 bloques o *tiles*, donde cada uno contiene 2 núcleos, 4 unidades de procesamiento vectorial (VPUs) (2 por núcleo) y una caché compartida L2 de 1 MB. Para la interconexión de todos los *tiles* y la coherencia de caché entre ellos, KNL utiliza una malla de interconexión 2D. Esta malla conecta *tiles*, controladores de memoria, controladores de E/S y otros módulos en el chip. En esta nueva iteración de la arquitectura MIC se utilizan dos tipos de memoria: memoria DRAM multicanal (MCDRAM) y memoria DDR. Además contiene 2 controladores de memoria DDR (DDRMC), y la comunicación con el *host* se realiza a través de un bus PCIe 3.0.

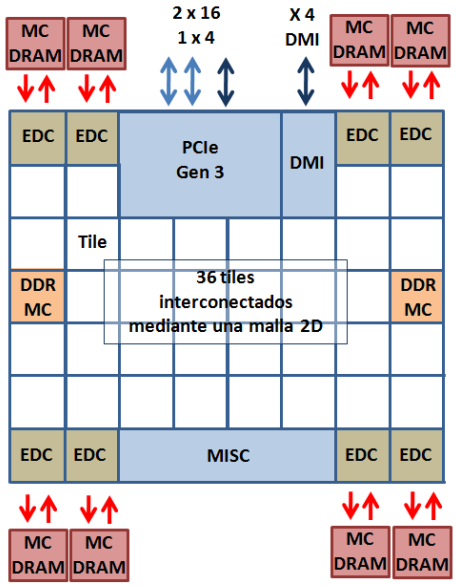


Figura 1.11: Microarquitectura MIC (many-integrated-core) Knights Landing KNL. KNL contiene 36 bloques o *tiles* interconectados mediante una malla 2D de coherencia de caché. Cada *tile* contiene 2 núcleos. Además proporciona controladores de memoria DRAM multicanal (MCDRAM) y memoria DDR (DDRMC) y soporte para bus PCIe 3.0.

1.2.3. Interfaces de Programación (APIs)

Aunque en los últimos años han aparecido varias interfaces o lenguajes de programación para las arquitecturas GPU y MIC, la mayoría se implementan sobre dos de los más habituales lenguajes de programación para estos entornos, como son NVIDIA-CUDA y OpenCL. En esta sección daremos una breve visión de estos dos lenguajes de programación centrándonos en el uso de sus principales funciones y como los diferentes recursos hardware son tratados e identificados por cada uno de ellos.

NVIDIA-CUDA

CUDA [59] son las siglas de *Compute Unified Device Architecture* (Arquitectura Unificada de Dispositivos de Cómputo) que hace referencia tanto a un

compilador como a un conjunto de herramientas de desarrollo creadas por NVIDIA, que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en GPUs de NVIDIA. En un entorno de programación CUDA hay tres abstracciones clave: 1) una jerarquía de grupos de hilos, 2) memorias compartidas y 3) barreras de sincronización. Estas abstracciones proporcionan paralelismo de datos de grano fino y paralelismo de hilos, alojados dentro de un paralelismo de datos de grano grueso y un paralelismo de tareas. Así, estas características permiten al programador dividir el problema en subproblemas que pueden ser resueltos independientemente en paralelo mediante bloques de hilos, y cada subproblema en partes más pequeñas que pueden ser resueltas cooperativamente mediante todos los hilos de un bloque.

Cada bloque de hilos puede ser planificado en cualquier multiprocesador (SM) de la GPU, en cualquier orden, concurrentemente o secuencialmente. Esto conlleva que un programa CUDA se puede ejecutar sobre cualquier número de multiprocesadores y por tanto no hay que limitarlo a una GPU específica. La Figura 1.12 muestra un ejemplo para un programa CUDA que lanza 8 bloques de hilos. Estos bloques son enviados a una GPU con 2 SMs y a una GPU con 4 SMs y cada bloque se ejecutaría en un SM. Si suponemos que cada bloque de hilos tarda el mismo tiempo en ejecutarse, la GPU con 4 SMs sería más rápida en realizar la computación. De esta manera, CUDA proporciona una escalabilidad automática sin ningún cambio en el software.

```
1  //Definicion del kernel
2  __global__ void vecAdd(float *A, float *B, float *C)
3  {
4      int i = threadIdx.x;
5      C[i] = A[i] + B[i]
6  }
7
8  int main()
9  {
10     //Invocacion del kernel
11     vecAdd<<<1, N>>>(A, B, C);
12 }
```

Código 1.1: Código CUDA para la suma de vectores con un bloque de hilos.

CUDA es una extensión del lenguaje C que permite al programador definir funciones de C llamadas *kernels*, que se ejecutan N veces en paralelo mediante N hilos CUDA diferentes. Un *kernel* se define usando el identificador reservado *__global__* y especificando el número de hilos CUDA que ejecutará el *kernel*. Cada hilo tiene un identificador único que es accesible a través de la variable *threadIdx*. El Código 1.1 presenta un ejemplo en CUDA que suma dos vectores A y B de tamaño N y almacena el resultado en un vector C. Este código lanza un *kernel*

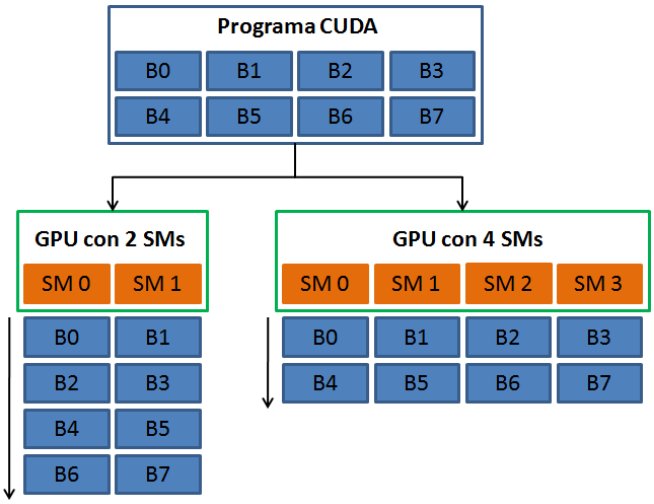


Figura 1.12: Una GPU está constituida por un conjunto de multiprocesadores (SMs). Un programa CUDA es particionado en bloques de hilos que se ejecutan de forma independiente, por lo que idealmente una GPU con más multiprocesadores puede ejecutar automáticamente el mismo programa CUDA en menos tiempo que una GPU con menos multiprocesadores.

que emplea un solo bloque de N hilos.

El número de hilos por bloque y el número de bloques se especifican mediante la sintaxis <<< ... >>> (línea 11 en el Código 1.1) a la cual se le puede pasar un entero (*int*) o el tipo tridimensional de CUDA *dim3*. El conjunto de bloques lanzados por un *kernel* se denomina *grid*. Tanto los bloques de hilos como el *grid* pueden tomar dimensiones 1D, 2D o 3D. Por lo tanto, cada bloque dentro del *grid* puede ser identificado por un índice 1D, 2D, o 3D accesible dentro del *kernel* a través de la variable $blockIdx$. De la misma forma, la dimensión del bloque de hilos es también accesible mediante la variable $blockDim$. El Código 1.2 muestra un ejemplo CUDA para la suma de dos matrices bidimensionales. En este ejemplo se emplean bloques bidimensionales de 256 hilos (16x16). El *grid* se crea con suficientes bloques de hilos como para tener un hilo por elemento de la matriz.

Los hilos dentro de un bloque pueden cooperar compartiendo datos a través de la memoria compartida. Los accesos a memoria compartida deben ser coordinados, por lo que es conveniente que los hilos de un bloque se sincronicen antes de acceder a la misma. La sincronización de los hilos de un bloque se consigue

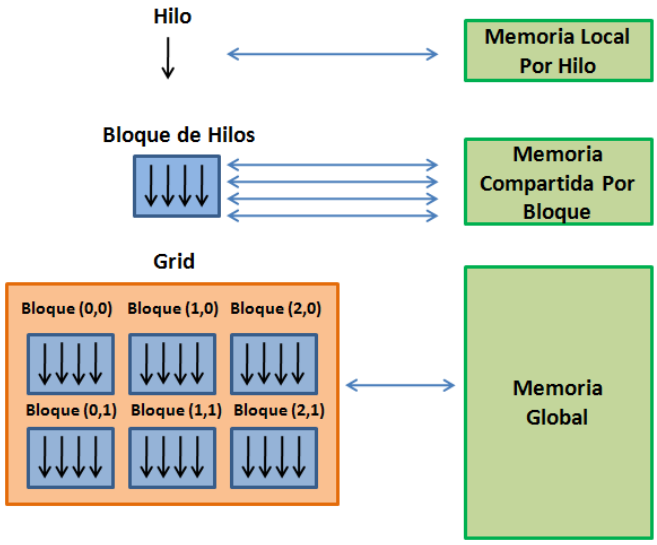


Figura 1.13: Jerarquía de Memoria en CUDA.

```
//Definicion del kernel
__global__ void matAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if(i < N && j < N)
        C[i][j] = A[i][j] + B[i][j]
}

int main()
{
    //Invocacion del kernel
    dim3 hilosPorBloque(16, 16);
    dim3 numBloques(N/hilosPorBloque.x, N/hilosPorBloque.y);

    matAdd<<<numBloques, hilosPorBloque>>>(A, B, C);
}
```

Código 1.2: Código CUDA para la suma de matrices.

mediante la función de CUDA *__syncthreads*(). Esta función actúa como una barrera en la cual todos los hilos del bloque deben esperar.

Los hilos CUDA acceden durante su ejecución a datos localizados en diferentes espacios de memoria como se muestra en la Figura 1.13. Cada hilo tiene un espacio de memoria local privada. Cada bloque de hilos tiene un espacio de memoria compartida visible para todos los hilos del bloque. Por último, todos los hilos tienen acceso a un espacio de memoria global de la GPU.

En un programa CUDA, la ejecución de una tarea típica en la GPU requiere la realización de una serie de pasos que se detallan en el Código 1.3:

1. Seleccionar el dispositivo CUDA (línea 5 del Código 1.3).
2. Reservar memoria en el *host* y en el dispositivo (líneas 10-16 del Código 1.3).
3. Transferir los datos de entrada desde el *host* al dispositivo (líneas 21-22 del Código 1.3).
4. Lanzar el *kernel* (líneas 25-28 del Código 1.3).
5. Transferir los datos de salida desde el dispositivo CUDA al *host* (línea 31 del Código 1.3).
6. Liberar memoria en el *host* y en el dispositivo (líneas 34-35 del Código 1.3).

En CUDA, todas las operaciones lanzadas a la GPU (*kernels* y transferencias de memoria) son ejecutadas mediante un *stream*. Un *stream* es una secuencia de operaciones lanzadas por el *host* que se ejecutan en orden en la GPU. Mientras que en las operaciones lanzadas en el mismo *stream* se garantiza su orden, las operaciones lanzadas en distintos *streams* pueden intercalar su ejecución e incluso ejecutarse concurrentemente si los recursos hardware en la GPU lo permiten. Cuando el programador no especifica ningún *stream*, se utiliza implícitamente el *stream* por defecto (*default stream*). El *default stream* es diferente a los demás *streams* porque se sincroniza con todas las operaciones lanzadas en la GPU. Por lo tanto, ninguna operación lanzada en el *default stream* comenzará hasta que todas las operaciones lanzadas previamente en la GPU hayan terminado, así como también ninguna operación en la GPU comenzará hasta que la última operación lanzada desde el *defaul stream* se haya completado[1]. Para el Código 1.3, CUDA utiliza implícitamente el *default stream* para lanzar los comandos, por lo tanto las operaciones de transferencias de memoria (*cudaMemcpy*) y *kernels* son ejecutados en la GPU en el orden en que han sido lanzadas. En el Código 1.3 desde el punto de vista del *host* las operaciones de transferencias de memoria

[1] A partir de CUDA 7 la sincronización con el default-stream puede ser evitada mediante la creación de los streams con el flag *cudaStreamNonBlocking*

```
1  int main()
2  {
3      int N = 1024;
4
5      cudaSetDevice(0);
6
7      float *h_A, *h_B, *h_C;
8      float *d_A, *d_B, *d_C;
9
10     //Reservar memoria en el host
11     h_A = (float *)malloc(N * N * sizeof(float));
12     h_B = (float *)malloc(N * N * sizeof(float));
13     //Reservar memoria en el dispositivo CUDA
14     cudaMalloc((void **)&d_A, N * N * sizeof(float));
15     cudaMalloc((void **)&d_B, N * N * sizeof(float));
16     cudaMalloc((void **)&d_C, N * N * sizeof(float));
17
18     //Inicializamos los datos en el host
19
20     //Transferimos los datos desde el host al dispositivo CUDA
21     cudaMemcpy(d_A, h_A, N * N * sizeof(float),
       cudaMemcpyHostToDevice);
22     cudaMemcpy(d_B, h_B, N * N * sizeof(float),
       cudaMemcpyHostToDevice);
23
24     //Lanzamos el kernel
25     dim3 hilosPorBloque(16, 16);
26     dim3 numBloques(N/hilosPorBloque.x,  N/hilosPorBloque.y);
27
28     matAdd<<<numBloques, hilosPorBloque>>>(d_A, d_B, d_C);
29
30     //Transferimos los datos desdel el dispositivo CUDA al host.
31     cudaMemcpy(h_C, d_C, N * N * sizeof(float),
       cudaMemcpyDeviceToHost);
32
33     //Liberamos memoria en el host y en el dispositivo CUDA
34     free(h_A);  free(h_B);  free(h_C);
35     cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
36
37 }
```

Código 1.3: Código CUDA para la suma de matrices. En este código se muestra todos los pasos previos a lanzar el kernel CUDA.

(*cudaMemcpy*) son síncronas ya que son realizadas con funciones bloqueantes, pero el *kernel* es asíncrono. Como la transferencia de memoria de la línea 22 en el Código 1.3 es una función síncrona, el *host* no lanzará el *kernel* hasta que esta

operación de memoria se haya completado. Una vez que el *kernel* es lanzado, el *host* pasa inmediatamente a la línea 31 en el Código 1.3, pero la operación de transferencia en esa línea no comenzará debido al orden impuesto por el *default stream* en la GPU.

Los *streams* definidos por el programador (*non-default streams*) son creados y destruidos por el *host*. Para lanzar una transferencia de memoria en un *non-default stream*, se puede utilizar la función no bloqueante de CUDA denominada *cudaMemcpyAsync*, que es similar a *cudaMemcpy* pero toma como quinto argumento un identificador de *stream*. De la misma manera, para lanzar un *kernel* en un *non-default stream* debemos pasar un identificador de *stream* como cuarto parámetro de ejecución (el tercer parámetro es la cantidad de memoria compartida, reservada dinámicamente por el *kernel*). El Código 1.4 muestra las funciones asíncronas CUDA utilizadas para lanzar operaciones de memoria y *kernels*. Aunque desde el punto de vista del *host*, las operaciones realizadas por las funciones del Código 1.4 son asíncronas, desde el punto de vista de la GPU son síncronas, debido a que son lanzadas por el mismo *non-default stream* (stream1 en este caso). No obstante, habrá situaciones donde tengamos que sincronizar la ejecución del *host* con las operaciones lanzadas en un *non-default stream*. La manera más simple es utilizar la función *cudaDeviceSynchronize*, donde el *host* se bloquea hasta que todas las operaciones lanzadas en la GPU hayan finalizado, como se muestra en la línea 19 del Código 1.4. Si, por el contrario, solo queremos comprobar de una manera no bloqueante para el *host*, si las operaciones lanzadas en un *non-default stream* han terminado, podríamos utilizar la función *cudaStreamQuery(stream)*.

CUDA también proporciona una forma tanto para monitorizar la ejecución de la GPU, como para obtener mediciones del tiempo de ejecución de diferentes comandos. Esta tarea es llevada a cabo por el *host* mediante la manipulación e inserción asíncrona de eventos CUDA. El *host* puede insertar un evento en cualquier punto del programa y consultar su estado. Un evento alcanza su estado de completado cuando todas las tareas, u opcionalmente todos los comandos precedentes al evento en un stream dado, se han completado. Tanto la creación como la destrucción de los eventos son realizados por el *host*. El Código 1.5 muestra como se crean y utilizan los eventos CUDA para la medición de tiempos de ejecución. En primer lugar se crean los eventos mediante la función *cudaEventCreate* (líneas 5-6, Código 1.5). A continuación se insertan los eventos de inicio y fin al principio y al final del bloque de código que se quiere medir (líneas 9 y 21, Código 1.5). La inserción de estos eventos se realiza en el *stream* 0 (*default stream*), mediante la función *cudaEventRecord*. Una vez insertados los eventos de inicio y fin, es conveniente sincronizar la ejecución del programa *host* mediante la función *cudaEventSynchronize* (línea 23, Código 1.5), y calcular el tiempo

transcurrido entre los eventos mediante la función *cudaEventElapsedTime* (linea 25, Código 1.5). Por último, se destruyen los eventos mediante la función *cudaEventDestroy* (lineas 28-29, Código 1.5).

```
1
2  //Estructura stream
3  cudaStream_t stream1;
4  //Creamos el stream
5  cudaStreamCreate(&stream1);
6
7
8  //Transferencias asincronas desde el host a la GPU.
9  cudaMemcpyAsync(d_A, h_A, N*N*sizeof(float),cudaMemcpyHostToDevice,
       stream1);
10 cudaMemcpyAsync(d_B, h_B, N*N*sizeof(float),cudaMemcpyHostToDevice,
       stream1);
11
12 //Lanzamos el kernel.
13 matAdd<<<numBloques , hilosPorBloque, 0, stream1>>>(d_A, d_B, d_C) ;
14
15 //Transferencia asincrona desde la GPU al host.
16 cudaMemcpyAsync(h_C, d_C, N*N*sizeof(float),cudaMemcpyDeviceToHost,
       stream1);
17
18 //Sincronizamos con la ejecucion en GPU
19 cudaDeviceSynchronize();
20
21 //Destruimos el stream
22 cudaStreamDestroy(stream1);
```

Código 1.4: Funciones CUDA para realizar operaciones de transferencia de memoria asíncronas mediante un *non-default stream.*

Como se ha comentado anteriormente, los eventos también proporcionan un mecanismo para saber cuando se ha producido una operación en un stream. La función *cudaEventSynchronize(event)* bloquea la ejecución del *host* hasta que el evento *event* ha alcanzado su estado de completado. Este comportamiento se puede observar en la línea 23 del Código 1.5, donde el *host* espera a que el evento *fin* se haya completado para después medir el tiempo transcurrido entre los eventos *ini* y *fin*. Si solamente se quiere consultar el estado de un evento sin bloquear el *host*, se debe usar la función *cudaEventQuery(event)*. Las operaciones lanzadas en un stream también pueden ser sincronizadas mediante eventos. El Código 1.6 es una modificación del Código 1.4, donde ahora se utilizan 3 streams para lanzar los comandos y se sincroniza su ejecución mediante eventos de forma que los comandos de cada stream no se ejecuten hasta que hayan terminado los del stream anterior. En primer lugar se crean los streams y eventos CUDA (líneas 2-13, Código 1.6), y se lanzan los comandos de transferencia desde el *host* a la

GPU en el stream *stream1* (líneas 16-17, Código 1.6), seguidos de la inserción del evento *evt1* en el mismo stream (línea 19, Código 1.6). En segundo lugar, se lanza el *kernel* por el stream *stream2*, pero su ejecución no debe empezar hasta que el último comando lanzado por el stream *stream1* se haya completado. Para ello, se inserta de forma asíncrona una barrera en el stream *stream2*, dependiente del evento *evt1*. La inserción de esta barrera se realiza mediante la función *cudaStreamWaitEvent(stream2, evt1)* (línea 22, Código 1.6). La barrera insertada por esta función hace que los siguientes comandos lanzados en el stream *stream2* no comiencen hasta que el evento *evt1* haya alcanzado el estado de completado. Esta sincronización se realiza de forma eficiente en la GPU. A continuación, se inserta en el stream *stream2* el evento *evt2* para monitorizar la ejecución del *kernel* (línea 27, Código 1.6). Por último, se lanza la transferencia desde la GPU al *host* por el stream *stream3*, pero insertando antes en este stream una barrera dependiente del evento *evt2* para sincronizarlo con el stream *stream2*. La inserción de esta barrera se realiza de forma similar a la realizada con el *kernel* (líneas 30-34, Código 1.6). Una vez lanzados todos los comandos en la GPU, el *host* debe esperar a que estos hayan terminado (línea 37, Código 1.6).

```
1
2  //Estructuras evento CUDA
3  cudaEvent_t ini, fin;
4  //Creamos los eventos
5  cudaEventCreate(&ini);
6  cudaEventCreate(&fin);
7
8  //Insertamos el evento ini en el stream 0 (non-default stream)
9  cudaEventRecord(ini, 0);
10
11 cudaMemcpyAsync(d_A, h_A, N*N*sizeof(float),cudaMemcpyHostToDevice,
       stream1);
12 cudaMemcpyAsync(d_B, h_B, N*N*sizeof(float),cudaMemcpyHostToDevice,
       stream1);
13
14 //Lanzamos el kernel.
15 matAdd<<<numBloques , hilosPorBloque, 0, stream1>>>(d_A, d_B, d_C) ;
16
17 //Transferencia asincrona desde la GPU al host.
18 cudaMemcpyAsync(h_C, d_C, N*N*sizeof(float),cudaMemcpyDeviceToHost,
       stream1);
19
20 //Insertamos el evento stop en el stream 0 (non-default stream)
21 cudaEventRecord(fin, 0);
22 //Esperamos a que el evento fin se haya completado
23 cudaEventSynchronize(fin);
24 float tiempo;
25 cudaEventElapsedTime(&tiempo, ini, fin);
26
```

```
//Destruimos los eventos.
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Código 1.5: Funciones CUDA para la medición del tiempo de ejecución de comandos en la GPU. Unos eventos CUDA son insertados al inicio y al final del bloque de código del cual queramos medir su tiempo de ejecución.

```
//Estructuras stream
cudaStream_t stream1, stream2, stream3;
//Creamos los streams
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaStreamCreate(&stream3);

//Estructuras evento CUDA
cudaEvent_t evt1, evt2, evt3;
//Creamos los eventos
cudaEventCreate(&evt1);
cudaEventCreate(&evt2);
cudaEventCreate(&evt3);

//Lanzamos comandos de transferencia en el stream1
cudaMemcpyAsync(d_A, h_A, N*N*sizeof(float),cudaMemcpyHostToDevice,
    stream1);
cudaMemcpyAsync(d_B, h_B, N*N*sizeof(float),cudaMemcpyHostToDevice,
    stream1);
//Insertamos el evento evt1 en el stream1
cudaEventRecord(evt1, stream1);

//Sincronizamos el stream2, con el evento evt1 (stream1)
cudaStreamWaitEvent(stream2, evt1);

//Lanzamos el kernel por el stream2
matAdd<<<numBloques , hilosPorBloque, 0, stream2>>>(d_A, d_B, d_C) ;
//Insertamos el evento evt2 en el stream2
cudaEventRecord(evt2, stream2);

//Sincronizamos el stream3, con el evento evt2 (stream2)
cudaStreamWaitEvent(stream3, evt2);
//Lanzamos comandos de transferencia en el stream3
cudaMemcpyAsync(h_C, d_C, N*N*sizeof(float),cudaMemcpyDeviceToHost,
    stream3);
//Insertamos el evento evt3 en el stream3
cudaEventRecord(evt3, stream3);

//Sincronizamos el host con el evento evt3
cudaEventSynchronize(evt3);

//Destruimos los eventos.
```


```
40 cudaEventDestroy(evt1);
41 cudaEventDestroy(evt2);
42 cudaEventDestroy(evt3);
43
44 //Destruimos los streams
45 cudaStreamDestroy(stream1);
46 cudaStreamDestroy(stream2);
47 cudaStreamDestroy(stream3);
```

Código 1.6: Funciones CUDA para la sincronización de streams mediante eventos CUDA.

En situaciones donde se requiere ejecutar de forma concurrente diferentes tareas en el acelerador, el uso de los streams es fundamental para conseguir este comportamiento. El Código 1.7 muestra un ejemplo explicativo de como sería la estructura del código CUDA para conseguir la ejecución concurrente de dos tareas en el acelerador. Para conseguir este comportamiento, cada tarea debe ser lanzada por un stream diferente. Por tanto, la computación de una tarea se podrá realizar concurrentemente con las transferencia de datos de otra tarea diferente. Supongamos que tenemos 2 tareas nombradas como T0, T1. Los datos en el *host* para las tareas T0 y T1 serán h_T0 y h_T1 respectivamente. De forma similar, los datos en el acelerador para las tareas T0 y T1 serán d_T0 y d_T1. Por simplicidad, no se han incluido la reserva de memoria e inicialización de los datos de las tareas T0 y T1 tanto en el *host* como en el acelerador, pero deberían hacerse antes de ejecutar el Código 1.7. En primer lugar debemos crear los streams para lanzar los comandos de las diferentes tareas (líneas 8-11, Código 1.7). A continuación, se deben lanzar las transferencias de memoria y los kernels de cada una de las tareas por streams diferentes. Lanzamos las transferencias de memoria y el kernel de la tarea T0 por el stream 0 (líneas 18-22, Código 1.7). Después lanzamos las transferencias de memoria y el kernel de la tarea T1 por stream 1 (líneas 26-30, Código 1.7). De esta manera, el kernel de la tarea T0 se podrá ejecutar concurrentemente con la transferencia *HtD* de la tarea T1 y, el kernel de la tarea T1 se ejecutará concurrentemente con la transferencia *DtH* de la tarea T0.

```
1
2 //Numero de tareas
3 int n_tareas = 2;
4 //Numero de streams
5 int n_streams = n_tareas;
6
7 //Creamos los streams
8 cudaStream_t *streams = new cudaStream_t[n_streams];
9
10 for(int i = 0; i < n_streams; i++)
11     cudaStreamCreate(&streams[i]);
```

```
12
13
14 //Lanzamos cada tarea por un stream diferente
15
16 //Lanzamos la tarea T0 por el stream 0
17 //Transferencia HtD
18 cudaMemcpyAsync(d_T0, h_T0, T0_bytesHtD, cudaMemcpyHostToDevice,
       streams[0]);
19 //Kernel
20 kernel_T0<<<numBloques_T0, hilosporBloque_T0, 0, streams[0](d_T0)
21 //Transferencia DtH
22 cudaMemcpyAsync(h_T0, d_T0, T0_bytesDtH, cudaMemcpyDeviceToHost,
       streams[0]);
23
24 //Lanzamos la tarea T1 por el stream 1
25 //Transferencia HtD
26 cudaMemcpyAsync(d_T1, h_T1, T1_bytesHtD, cudaMemcpyHostToDevice,
       streams[1]);
27 //Kernel
28 kernel_T0<<<numBloques_T1, hilosporBloque_T1, 0, streams[1](d_T1)
29 //Transferencia DtH
30 cudaMemcpyAsync(h_T1, d_T1, T1_bytesDtH, cudaMemcpyDeviceToHost,
       streams[1]);
31
32 //Sincronizamos el host con el acelerador
33 cudaDeviceSynchronize();
```

Código 1.7: Solapamiento de computación y transferencias de dos tareas diferentes en CUDA.

OpenCL

OpenCL [34] (Open Computing Language, en español lenguaje de computación abierto) consta de una interfaz de programación de aplicaciones y de un lenguaje de programación que permiten crear aplicaciones con paralelismo a nivel de datos y de tareas. Estas aplicaciones pueden ejecutarse tanto en unidades centrales de procesamiento como en cualquier tipo de acelerador (GPUs, arquitecturas MIC, FPGAs, etc). El lenguaje está basado en C99, eliminando cierta funcionalidad y extendiéndolo con operaciones vectoriales. Apple creó la especificación original y fue desarrollada en conjunto con AMD, IBM, Intel y NVIDIA. Apple la propuso al Grupo Khronos para convertirla en un estándar abierto y libre de derechos. El 16 de junio de 2008 Khronos creó el *Compute Working Group* para llevar a cabo el proceso de estandarización.

OpenCL soporta un amplio rango de aplicaciones. Una generalización de todas las aplicaciones soportadas por OpenCL sería bastante complejo de exponer. Sin

embargo, una aplicación en OpenCL para plataformas heterogéneas debe realizar los siguientes pasos:

1. Descubrir los componentes que contiene la plataforma heterogénea.
2. Comprobar las características de estos componentes, de tal manera que el software pueda adaptarse a las características hardware específicas de cada uno de ellos.
3. Crear los bloques de instrucciones (*kernels*) que se ejecutarán en la plataforma.
4. Instanciar y manipular los objetos de memoria involucrados en la computación.
5. Ejecutar tanto los *kernels* como los componentes necesarios en el orden correcto.
6. Recoger los resultados finales.

Todos estos pasos se pueden agrupar en los siguientes modelos:

- **Modelo de la Plataforma:** Una descripción a alto nivel del sistema heterogéneo.
- **Modelo de Ejecución:** Una representación abstracta de cómo se ejecutarán las distintas instrucciones en la plataforma.
- **Modelo de Memoria:** El conjunto de regiones de memoria dentro de la aplicación OpenCL y como interactúan durante la computación.

Modelo de la Plataforma. El modelo de la plataforma OpenCL es una representación a alto nivel de cualquier plataforma heterogénea utilizada. Este modelo se muestra en la Figura 1.14. Una plataforma OpenCL siempre incluye un único *host* que interactúa con el entorno externo al programa OpenCL, incluyendo E/S o cualquier interacción con otros programas de usuario. El *host* se conecta a uno o más dispositivos OpenCL, los cuales se dividen en unidades de cómputo (CUs) y estas, a su vez, en elementos de procesamiento (PEs). El Código 1.8 muestra las distintas funciones OpenCL necesarias para obtener la descripción de la plataforma.

```
1
2     cl_platform_id plataforma;   //Plataforma OpenCL
3     cl_device_id dispositivo;    //Dispositivo OpenCL
4
5     //Descripcion de la plataforma
6     clGetPlatformIDs(1, &plataforma, NULL);
7     //ID dispositivo OpenCL (GPU)
8     clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &dispositivo,
      NULL);
```

Código 1.8: Código OpenCL para obtener la descripción de la plataforma

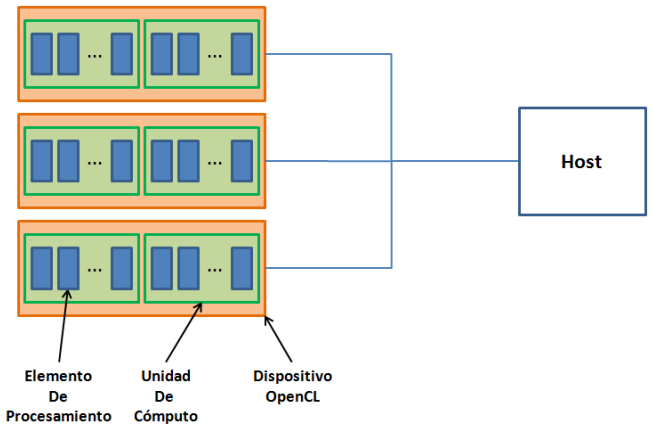


Figura 1.14: Modelo de plataforma OpenCL con un host y uno o más dispostivos OpenCL. Cada dispositivo OpenCL tiene uno o más unidades de cómputo (CU) y cada una de ellas uno o más elementos de procesamiento (PE).

Modelo de Ejecución. Una aplicación OpenCL consta de dos partes: el *programa del host* y un conjunto de uno o más *kernels*. Los *kernels* se ejecutan en los dispositivos OpenCL y son simples funciones que transforman los objetos de memoria de entrada en objetos de memoria de salida. Un *kernel* OpenCL se escribe en el lenguaje de programación OpenCL C y es compilado en tiempo de ejecución por el compilador OpenCL. Más precisamente, el modelo de ejecución describe como se ejecutan los *kernels* en los dispositivos OpenCL. El *host* define un *kernel* y lo envía a un dispositivo OpenCL para su ejecución, creando un espacio de N instancias del *kernel* denominado *NDRange*. Cada instancia de un *kernel* se denomina *workitem* y es identificada por su coordenadas (1D, 2D o 3D) dentro del espacio *NDRange*. Estas coordenadas son el identificador principal (ID) del *workitem*. Los *workitems* se agrupan en *wavefronts*. Un *wavefront* es un conjunto de 64 *workitems* que se ejecuta en una CU. Los *wavefront* se agrupan

en *workgroups*, los cuales tienen un identificador principal definido en el mismo espacio que los *workitems*. Dentro de un *workgroup* cada *workitem* tiene su identificador local y se ejecutan de forma concurrente. La Figura 1.15 representa un espacio 2D de N instancias del *kernel*, donde las instancias se reparten en 4x4 *workgroups* y estos a su vez se dividen en 5x5 *workitems*. La Tabla 1.2 muestra para el modelo de ejecución, la equivalencia entre las terminologías usadas por programadores CUDA y programadores OpenCL.

CUDA	OpenCL
Grid	NDRange
Thread Block	Workgroup
Warp	Wavefront
Thread	Workitem
Thread ID	Global ID
Block Index	Block ID
Thread Index	Local ID

Tabla 1.2: Equivalencia entre las terminologías usadas en entornos CUDA y OpenCL para el modelo de ejecución.

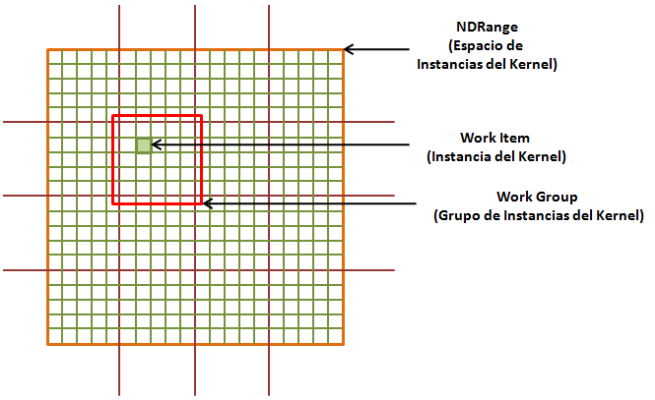


Figura 1.15: Ejemplo de espacio 2D de N instancias del *kernel*. Las instancias se reparten en 4x4 *workgroups* (recuadro rojo) y cada *workgroup* está compuesto por 5x5 instancias del kernel (recuadro verde).

Antes de que se defina el espacio *NDRange* y el *kernel* sea enviado al dispositivo OpenCL, el *host* tiene que establecer el contexto OpenCL para los *kernels*, así como las colas que controlan los detalles de cómo y cuándo el *kernel* se ejecutará. El contexto OpenCL define el entorno dentro del cual los *kernels* se definen y se ejecutan. Este es creado y manipulado por el *host*. La interacción entre el *host* y

los dispositivos OpenCL se lleva acabo mediante comandos que son introducidos por el *host* en una cola de comandos. Una cola de comandos es creada por el *host* y asociada a un único dispositivo OpenCL después de que el contexto se haya definido. Los comandos esperan en la cola de comandos a que sean enviados al dispositivo para su ejecución y pueden ser de tres tipos:

- Comandos de Ejecución de Kernel. Estos comandos ejecutan un *kernel* en los elementos de procesamiento del dispositivo.
- Comandos de Memoria. Este tipo de comandos transfieren datos entre el *host* y los distintos objetos de memoria, o entre objetos de memoria, y mapean y des-mapean objetos del espacio de memoria del *host*.
- Comandos de Sincronización. Los comandos de sincronización ponen restricciones en el orden de ejecución de los otros comandos de la cola.

Las colas de comandos se pueden definir en dos modos:

- En Orden. Los comandos son lanzados y ejecutados en el orden en que aparecen en la cola de comandos.
- Fuera de Orden. Los comandos son lanzados en el orden en que aparecen en la cola de comandos, pero no se garantiza que se ejecuten en ese mismo orden en el dispositivo. Este tipo de configuración de la cola sería útil cuando se quiere ejecutar concurrentemente comandos que son totalmente independientes entre si.

```
1 //Definicion del kernel
2 __kernel void matadd(__global float* A, __global float* B, __global
      float* C) {
3
4     int i = get_global_id(0);
5     C[i] = A[i] + B[i];
6  }
```

Código 1.9: Kernel OpenCL para la suma de matrices.

Finalmente, es posible asociar varias colas de comandos a un contexto para cualquier dispositivo OpenCL. Estas colas se ejecutarían concurrente e independientemente sin ningún mecanismo implícito de sincronización entre ellas. El Código 1.9 muestra un ejemplo de *kernel* OpenCL donde se realiza la suma de dos matrices. El *kernel* se define mediante el tipo reservado *__kernel*. Las matrices de entrada A y B y la matriz de salida C, son vectores definidos en memoria global

del dispositivo OpenCL, por lo tanto se definen con el tipo reservado *__global*. A continuación, se obtiene el índice global del *workitem* (línea 4), que servirá para identificar el elemento de las matrices de entrada y de salida que interviene en la computación.

Una vez definido el *kernel*, se tiene que crear el modelo de ejecución. El Código 1.10 muestra las funciones OpenCL necesarias para la creación del modelo de ejecución. En primer lugar se crea el contexto OpenCL (línea 10, Código 1.10) para a continuación crear la cola de comandos (línea 13, Código 1.10). La cola de comandos, como ya hemos comentado anteriormente, tendrá los comandos que el *host* enviará al dispositivo OpenCL. A continuación, se crea el programa OpenCL que contendrá el *kernel* OpenCL. Por lo tanto, se lee el archivo del *kernel* OpenCL y se compila en tiempo de ejecución (líneas 16-36, Código 1.10). Por último, una vez compilado el *kernel*, se obtiene una estructura de información del mismo para su interacción con él (linea 39, Código 1.10).

```
1
2     cl_context contexto;
3     cl_command_queue cola;
4     cl_program programa;
5     cl_kernel kernel;
6     cl_int err;
7
8     //Creacion del Contexto
9     //El dispositivo ha sido creado antes con el modelo de la
      plataforma
10    contexto = clCreateContext(NULL, 1, &dispositivo, NULL, NULL, &
      err);
11
12    //Creamos la cola de comandos para la interaccion con el
      dispositivo OpenCL
13    cola = clCreateCommandQueue(contexto, dispositivo, 0,&err);
14
15    //Leemos el fichero del kernel OpenCL
16    FILE *fich_programa;
17    char *programa_buffer, *programa_log;
18    size_t programa_tam, log_size;
19
20    fich_programa = fopen("matadd.cl", "r");
21    fseek(fich_programa, 0, SEEK_END);
22    program_tam = ftell(fich_programa);
23    rewind(fich_programa);
24    programa_buffer = (char*) malloc(program_tam + 1);
25    programa_buffer[program_tam] = '\0';
26    fread(programa_buffer, sizeof(char), program_tam, fich_programa)
      ;
27    fclose(fich_programa);
28
29    //Creamos el kernel OpenCL
30    programa = clCreateProgramWithSource(
31    contexto, 1, (const char**) &programa_buffer, &program_tam, &err
      );
32
33    free(program_buffer);
34
35    //Compilamos el kernel OpenCL
36    err = clBuildProgram(programa, 0, NULL, NULL, NULL, NULL);
37
38    //Creamos el kernel
39     kernel = clCreateKernel(programa, "matadd", &err);
```

Código 1.10: Código OpenCL para crear el modelo de ejecución en OpenCL para la suma de matrices. El código crea el contexto, cola de comandos, así como también, crea el programa OpenCL y lo compila en tiempo de ejecución.

Modelo de Memoria. El modelo de memoria define los detalles y tipos de objetos de memoria definidos en el contexto, así como también las reglas de como usarlos. OpenCL define dos tipos de objetos de memoria: objetos buffer y objetos imagen. Los objetos buffer son bloques contiguos de memoria disponibles para los *kernels*. Un programador puede mapear estructuras de datos en estos buffers y acceder a ellos mediante punteros. Por el contrario, los objetos imagen se restringen únicamente al almacenamiento de imágenes, aunque pueden ser 1D, 2D o 3D con uno o varios canales. El modelo de memoria define cinco regiones de memoria distintas:

- **Memoria del Host:** Esta región de memoria es solo visible por el *host*.
- **Memoria Global:** Esta región de memoria permite accesos de lectura/escritura por todos los *workitems* dentro de un *workgroup*.
- **Memoria Constante:** Esta región de memoria de la memoria global permanece constante durante la ejecución del *kernel*. Los *workitems* tiene solamente acceso de lectura a este tipo de memoria.
- **Memoria Local:** Esta región de memoria es local a un *workgroup* y puede ser usada para reservar variables compartidas por todos los *workitems* dentro del *workgroup*.
- **Memoria Privada:** Este tipo de memoria es privada a un *workitem*.

CUDA	OpenCL
Memoria del Host	Memoria del Host
Memoria Global	Memoria Global
Memoria Constante	Memoria Constante
Memoria Compartida	Memoria Local
Registros	Memoria Privada

Tabla 1.3: Equivalencia entre las terminologías usadas en entornos CUDA y OpenCL para el modelo de memoria.

La Tabla 1.3 muestra la equivalencia para el modelo de memoria, entre la terminología usada por programadores CUDA y OpenCL. La Figura 1.16 muestra como las distintas regiones de memoria relacionan el modelo de la plataforma con el modelo de ejecución. Los *workitems* se ejecutan en los elementos de procesamiento *PEs* y tienen su propia memoria privada. Un *workgroup* se ejecuta en una unidad de cómputo (*Compute Unit* o CU).y comparte una región de memoria local con los *workitems* del *workgroup*. El dispositivo OpenCL trabaja con el *host*

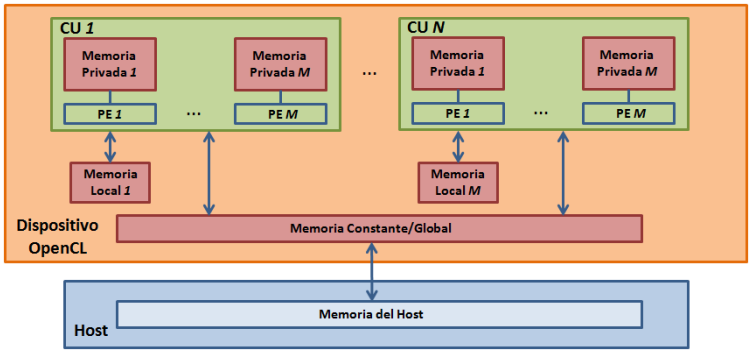


Figura 1.16: Resumen del modelo de memoria en OpenCL y como las distintas regiones de memoria iteractúan entre si.

por medio de la memoria global. Tanto el *host* como el dispositivo OpenCL son independientes uno del otro, pero su interacción es necesaria para la ejecución de un programa OpenCL. Esta interacción puede ocurrir de dos maneras: copiando datos explícitamente o mapeando y des-mapeando regiones de un objeto de memoria. Para copiar explícitamente datos, el *host* introduce, en la cola de comandos, comandos de transferencias de datos entre el objeto de memoria y la memoria del *host*. Estos comandos de transferencias pueden ser bloqueantes o no bloqueantes con respecto a la ejecución del *host*. Los métodos de mapeo/des-mapeo permiten al *host* mapear regiones de un objeto de memoria dentro de su propio espacio de direcciones. El comando de mapeo de memoria (el cual es encolado también en la cola de comandos) puede ser a su vez bloqueante o no bloqueante.

```
1
2     //Objetos de memoria en el dispositivo
3     cl_mem mat_A_buff, mat_B_buff, mat_C_buff;
4
5     //Matrices en el host
6     float mat_A[16];
7     float mat_B[16];
8     float mat_C[16] {10.0};
9
10    //Inicializacion de datos en el host
11    for(int i = 0; i < 16; i++)
12    {
13        mat_A[i] = 1.0;
14        mat_B[i] = 2.0;
15    }
16
```

```
17      //Creamos los objetos de memoria en el dispositivo
18       mat_A_buff = clCreateBuffer
19      (
20          contexto,
21          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
22          sizeof(float)*16,
23          mat_A,
24          &err
25      );
26
27      mat_A_buff = clCreateBuffer
28      (
29          contexto,
30          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
31          sizeof(float)*16,
32          mat_B,
33          &err
34      );
35
36
37      mat_C_buff = clCreateBuffer
38      (
39          contexto,
40          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
41          sizeof(float)*16,
42          mat_C,
43          &err
44      );
```

Código 1.11: Código OpenCL para crear el modelo de memoria necesario para la ejecución de la suma de matrices.

El Código 1.11 muestra la creación en OpenCL del modelo de memoria para la suma de matrices. En este código se crean los objetos de memoria necesarios (líneas 18-44) para la interacción con el modelo de ejecución presentado en el Código 1.10.

Una vez creada la descripción de la plataforma (Código 1.8), el modelo de ejecución (Código 1.10) y el modelo de memoria (Código 1.11), se pueden enviar comandos de transferencia de datos y computación al dispositivo OpenCL. El Código 1.12 muestra como se realiza el envío de comandos al dispositivo en OpenCL. En primer lugar se le pasan al *kernel* sus argumentos (líneas 3-5, Código 1.12). A continuación, se configura el espacio *NDRange* (líneas 9-11, Código 1.12) y se encolan los comandos de transferencia desde el *host* al dispositivo (líneas 14-15, código 1.12). En este código, desde el punto de vista del *host*, las funciones de transferencia de memoria (*clEnqueueWriteBuffer* y *clEnqueueReadBuffer*) son síncronas ya que son funciones bloqueantes debido al parámetro CL_TRUE. Sin

embargo, la función de envío del *kernel* (*clEnqueueNDRangeKernel*) es una función no bloqueante y asíncrona. Como las funciones *clEnqueueWriteBuffer* están definidas como funciones bloqueantes, el *host* no lanzará el *kernel* hasta que estas funciones de transferencia se hayan completado en el dispositivo. Una vez el *kernel* ha sido lanzado (línea 18, Código 1.12), el *host* envía inmediatamente la transferencia de vuelta desde el dispositivo (línea 21, Código 1.12). Esta transferencia es síncrona con respecto al *host* (parámetro CL_TRUE) y con respecto al dispositivo debido a que es lanzada por la misma cola de comandos que el *kernel*. Para lanzar transferencias de memoria no bloqueantes se debe pasar CL_FALSE como tercer parámetro a las funciones *clEnqueueWriteBuffer* y *clEnqueueReadBuffer*.

```
1
2    //Pasamos al kernel sus argumentos
3   clSetKernelArg(kernel,0,sizeof(cl_mem),&mat_A_buff);
4   clSetKernelArg(kernel,1,sizeof(cl_mem),&mat_B_buff);
5   clSetKernelArg(kernel,2,sizeof(cl_mem),&mat_C_buff);
6
7
8   //Configuramos el espacion NDRange
9     cl_uint dimensiones = 1;
10    size_t global_tam = 16;
11    size_t local_tam = 16;
12
13    //Comandos de tranferencia desde el host al dispositivo
14    clEnqueueWriteBuffer(cola,mat_A_buff,CL_TRUE,0,sizeof(float)*16,
      mat_A,0,NULL,NULL);
15    clEnqueueWriteBuffer(cola,mat_B_buff,CL_TRUE,0,sizeof(float)*16,
      mat_B,0,NULL,NULL);
16
17    //Encolamos el kernel
18    clEnqueueNDRangeKernel(cola,kernel,dimensiones,NULL,&global_tam
      ,&local_tam,0,NULL,NULL);
19
20    //Comando de transferencia desde el dipositivo al host
21    clEnqueueReadBuffer(cola,mat_C_buff,CL_TRUE,0,sizeof(float)*16,
      mat_C,0,NULL,NULL);
```

Código 1.12: Código OpenCL para el envío de comandos de la suma de matrices.

Aunque es posible sincronizar la ejecución del *host* con el progreso de los comandos en el dispositivo mediante funciones bloqueantes, habrá también situaciones en las que resulte interesante realizar tal sincronización con respecto a funciones no bloqueantes. La función *clFinish(CQ)* bloquea el progreso de la ejecución del *host* hasta que todas las operaciones lanzadas en la cola de comandos CQ se hayan completado en el dispositivo. OpenCL también dispone de eventos asociados a los comandos. La función *clGetEventInfo* devuelve información sobre

el estado de un evento OpenCL. Por lo tanto, el *host* puede saber si un comando ha sido ejecutado en el dispositivo, consultando el estado de su evento asociado. La sincronización del *host* también se puede conseguir mediante la función *clWaitForEvents*, donde este se bloquea hasta que una lista de eventos alcancen su estado de completado. El Código 1.13 muestra como se asocian eventos a los comandos lanzados al dispositivo. El evento asociado a un comando no alcanzará su estado de completado, hasta que su comando asociado no haya terminado su ejecución.

Las operaciones lanzadas en distintas colas de comandos también pueden ser sincronizadas entre si. Al igual que se puede conocer el progreso de un comando con los eventos OpenCL, estos también pueden servir para definir dependencias entre comandos de distintas colas. En el Código 1.14 se definen dependencias entre comandos lanzados en diferentes colas de comandos. En la línea 9 se lanza asíncronamente un comando de transferencia al que se le asocia el evento *evt1*. Este evento sirve como dependencia para el siguiente comando lanzado en la línea 10. El evento *evt1* es pasado a esta función como octavo parámetro, definiendo así una dependencia con el comando lanzado en la línea 9. De la misma manera, se definen las dependencias para el *kernel* (línea 13) y la transferencia de vuelta al *host* (línea 16).

```
1   //Eventos OpenCL
2   cl_event evt1, evt2, evt3, evt4;
3
4   //Comandos de tranferencia desde el host al dispositivo
5   clEnqueueWriteBuffer(cola,mat_A_buff,CL_FALSE,0,sizeof(float)
    *16,mat_A,0,NULL,&evt1);
6   clEnqueueWriteBuffer(cola,mat_B_buff,CL_FALSE,0,sizeof(float)
    *16,mat_B,0,NULL,&evt2);
7
8   //Encolamos el kernel
9   clEnqueueNDRangeKernel(cola,kernel,dimensiones,NULL,&global_tam
    ,&local_tam,0,NULL,&evt3);
10
11  //Comando de transferencia desde el dipositivo al host
12  clEnqueueReadBuffer(cola,mat_C_buff,CL_FALSE,0,sizeof(float)*16,
    mat_C,0,NULL,&evt4);
```

Código 1.13: Código OpenCL donde se asocian eventos a comandos lanzados al dispositivo. El *host* puede consultar el estado de estos eventos para saber si un comando se ha completado.

```
1
2   //Definimos 4 colas de comandos
3   cl_command_queue cqs[4];
4   //Eventos OpenCL
```

```
5     cl_event evt1, evt2, evt3, evt4;
6
7     //Lanzamos por cada cola de comandos una operacion
8
9     clEnqueueWriteBuffer(cqs[0],mat_A_buff,CL_FALSE,0,sizeof(float)
      *16,mat_A,0,NULL,&evt1);
10    clEnqueueWriteBuffer(cqs[1],mat_B_buff,CL_FALSE,0,sizeof(float)
      *16,mat_B,0,&evt1,&evt2);
11
12    //Encolamos el kernel
13    clEnqueueNDRangeKernel(cqs[2],kernel,dimensiones,NULL,&
      global_tam,&local_tam,0,&evt2,&evt3);
14
15    //Comando de transferencia desde el dipositivo al host
16    clEnqueueReadBuffer(cqs[3],mat_C_buff,CL_FALSE,0,sizeof(float)
      *16,mat_C,0,&evt3,&evt4);
```

Código 1.14: Código OpenCL donde definen dependencias entre comandos lanzados en distintas colas de comandos.

En OpenCL también es posible solapar la computación de una tarea con las transferencias de memoria de otra tarea diferente. Para ello, se deben lanzar los comandos de transferencias *HtD*, *DtH* y los kernels por colas de comandos diferentes. Concretamente para un acelerador con dos motores DMA, las transferencias *HtD* se deben lanzar por colas pares, las transferencias *DtH* por colas impares y los kernels por otra cola distinta. Este hecho se explicará con más detalle en la sección 2.2.2. El Código 1.15 muestra un ejemplo explicativo de como sería la estructura del código OpenCL, para conseguir la ejecución concurrente de dos tareas en un acelerador con dos motores DMA. Los comandos de una misma tarea serán lanzados por colas de comandos distintas, por lo que se necesitará la inserción de eventos de sincronización para cumplir las dependencias entre comandos de una misma tarea. De esta forma, la computación de una tarea se podrá realizar concurrentemente con las transferencia de datos de otra tarea diferente. Supongamos que tenemos 2 tareas nombradas como T0 y T1. Los datos en el *host* para las tareas T0 y T1 serán h_T0 y h_T1 respectivamente. De forma similar, los datos en el acelerador para las tareas T0 y T1 serán d_T0 y d_T1. Por simplicidad, en el Código 1.15 no se han incluido la creación del contexto OpenCL, reserva de memoria en el host y en el acelerador, inicialización de los datos, así como la creación del programa OpenCL y del kernel. En primer lugar, crearemos 3 colas de comandos para lanzar los comandos de las diferentes tareas (líneas 12-17, Código 1.15). A continuación, se crean los eventos de sincronización para cumplir las dependencias entre los comandos de una tarea (líneas 24-30, Código 1.15). Después, lanzamos los comandos de la tarea 0 (líneas 36-42, Código 1.15) debidamente asociados a sus eventos de sincronización. Una vez lan-

zados los comandos de la tarea 0, lanzamos los comandos de la tarea T1 (líneas 46-52, Código 1.15). De esta forma, el kernel de la tarea T0 se podrá ejecutar concurrentemente con la transferencia *HtD* de la tarea T1 y, el kernel de la tarea T1 se ejecutará concurrentemente con la transferencia *DtH* de la tarea T0.

```
1
2  /* Creamos el Contexto OpenCL ...*/
3
4  /* Creamos el dispositivo OpenCL...*/
5
6  cl_int err;
7
8  //Numero de tareas
9  int n_tareas = 2;
10
11 //Creamos 3 colas de comandos
12 int n_colas = 3;
13
14 cl_command_queue colas[n_colas];
15
16 for(int i = 0; i < n_colas; i++)
17      colas[i] = clCreateCommandQueue(contexto, dispositivo, 0,&err);
18
19 /* Creamos el programa OpenCL... */
20
21 /* Leemos, compilamos y creamos el kernel OpenCL para cada una de
       las tareas */
22
23 //Eventos de sincronizacion para las transferencias HtD de las
       tareas
24 cl_event evt_htd[n_tareas];
25
26 //Eventos de sincronizacion para las transferencias DtH de las
       tareas
27 cl_event evt_dth[n_tareas];
28
29 //Eventos de sincronizacion para los kernels de las tareas
30 cl_event evt_kernel[n_tareas];
31
32 //Lanzamos las transferencias HtD por la cola 0, las transferencias
       DtH por la cola 1 y los kernels por la cola 2
33
34 //Tarea 0
35 //transferencia HtD por la cola 0
36 clEnqueueWriteBuffer(colas[0], d_T0,CL_FALSE,0, T0_bytesHtD, h_T0,0,
       NULL,&evt_htd[0]);
37
38 //kernel por la cola 2
39 clEnqueueNDRangeKernel(colas[2],kernel_T0,dimensiones_T0,NULL,&
       global_tam_T0,&local_tam_T0,1,&evt_htd[0],&evt_kernel[0]);
```


```
40
41 //transferencia DtH en la cola 1
42 clEnqueueReadBuffer(colas[1],d_T1,CL_FALSE,0,T0_bytesDtH, h_T0,1,&
       evt_kernel[0],&evt_dth[0]);
43
44 //Tarea 1
45 //transferencia HtD por la cola 0
46 clEnqueueWriteBuffer(colas[0], d_T1,CL_FALSE,0, T1_bytesHtD, h_T1,0,
       NULL,&evt_htd[1]);
47
48 //kernel por la cola 2
49 clEnqueueNDRangeKernel(colas[2],kernel_T1,dimensiones_T1,NULL,&
       global_tam_T1,&local_tam_T1,1,&evt_htd[1],&evt_kernel[1]);
50
51 //transferencia DtH en la cola 1
52 clEnqueueReadBuffer(colas[1],d_T1,CL_FALSE,0,T1_bytesDtH, h_T1,1,&
       evt_kernel[1],&evt_dth[1]);
53
54 //Sincronizamos el host con el acelerador
55 for(int i = 0; i < n_colas; i++)
56     clFinish(colas[i]);
```

Código 1.15: Solapamiento de computación y transferencias de dos tareas diferentes en OpenCL.

1.3. Objetivos de la Tesis

A la luz del ejemplo motivador de la Sección 1.1, en esta tesis nos hemos planteado estudiar cómo se ejecuta un grupo de tareas en un acelerador, y de qué forma podemos mejorar la planificación de esas tareas para disminuir el tiempo total de ejecución y aumentar la ocupación de los recursos hardware. Para ello nos hemos marcado los siguientes objetivos:

- Obtener un modelo de ejecución de tareas que permita simular con exactitud los procesos de transferencias de datos y ejecución de kernels.
- Diseñar un sistema de ejecución de tareas que permita explotar las posibilidades de concurrencia entre comandos pertenecientes a tareas distintas.
- Proponer métodos que permitan encontrar, en tiempo de ejecución, cuál es el orden de planificación más adecuado para un grupo de tareas arbitrario.
- Generalizar todas las propuestas para que sean viables tanto sobre arquitecturas distintas (NVIDIA, AMD e Intel) como sobre entornos diferentes (CUDA y OpenCL).

- Validar todas las propuestas usando tareas reales y con cargas de trabajo heterogéneas que permitan representar cualquier combinación de grupos de tareas.

1.4. Estructura de la Tesis

En el Capítulo 2 se analizan los elementos que intervienen en la ejecución de una tarea en un acelerador, y se discuten los métodos y requisitos necesarios para ejecutar concurrentemente un grupo de tareas. Al inicio del capítulo se hace un repaso al estado del arte de los modelos de ejecución concurrente de tareas en un acelerador, discutiendo algunos de los trabajos más relevantes que analizan las transferencias entre el *host* y los aceleradores, la ejecución de kernels sobre dichos aceleradores y la ejecución concurrente de comandos. A continuación se analizan y comparan varias alternativas para reservar la memoria, y se discute como los comandos asíncronos para transferencias y ejecución de los kernels deben ser lanzados para conseguir un solapamiento entre los mismos. Después se presenta un modelo simple de ejecución de kernels que se usa habitualmente en la literatura, y otro de transferencias en el que hemos incluido una serie de modificaciones que nos permiten modelar las transferencias de forma más exacta. Todas las aportaciones anteriores se combinan para diseñar un modelo que permite simular la ejecución de un grupo de tareas y obtener una estimación del tiempo total de ejecución. Por último, el modelo de ejecución es validado con unos benchmarks que se han seleccionado para diseñar cargas de trabajo representativas, tanto en entornos CUDA como OpenCL.

En el Capítulo 3 se discute como planificar, en tiempo de ejecución, un grupo de tareas con el objetivo de minimizar el tiempo total de ejecución. Al igual que en el capítulo anterior, primero se hace un repaso de algunos de los trabajos más relevantes en el campo de la planificación dinámica de tareas en aceleradores. A continuación, se presenta un sistema de ejecución de grupos de tareas que permite analizar las tareas que se van a ejecutar en el acelerador y predecir su comportamiento usando el modelo de ejecución que se presentó en el capítulo anterior. Este sistema permite también forzar un orden de ejecución por lo que a continuación se discute una heurística de planificación en tiempo de ejecución capaz de establecer un orden de ejecución de tareas muy cercano al óptimo posible, reduciendo así el tiempo total de ejecución. Por último, el sistema de ejecución y la heurística son validados, para CUDA y OpenCL, mediante una serie de experimentos.

El Capítulo 4 aborda el problema de planificar un grupo de tareas desde un punto de vista teórico, haciendo uso de la Teoría de Planificación desarrollada

dentro del campo de la Investigación Operativa. Para ello se hace primero una revisión de algunos conceptos básicos sobre Teoría de la Planificación y como se pueden aplicar al caso particular de ejecución de tareas en un acelerador. Este enfoque distinto permite expresar este caso como un problema de tipo *Flow Shop* para el cual ya existen en la literatura diversas soluciones que serán discutidas a continuación, junto con una nueva solución que combina una de las soluciones aportadas por la Teoría de Planificación con nuestro modelo de ejecución de tareas. Por último, la validez de esta nueva solución es analizada con una serie de experimentos.

En el último capítulo se hace un resumen de las aportaciones más importantes que se han hecho en esta tesis junto con las publicaciones a las que han dado lugar. También se exponen una serie de líneas de investigación futuras que esperamos permitirán ampliar la aplicabilidad del modelo de ejecución de grupos de tareas y de las heurísticas de planificación presentadas.

2 Ejecución Concurrente de Tareas

En este capítulo se analizan los elementos que intervienen en la ejecución de una tarea en un acelerador, y se discuten los métodos y requisitos necesarios para ejecutar concurrentemente un grupo de tareas.

La ejecución de una tarea implica la ejecución de las distintas etapas que componen una tarea. Una tarea esta compuesta por tres etapas: transferencia de datos desde la CPU al acelerador (HtD), ejecución del kernel (K) y transferencia de los resultados de la computación desde el acelerador a la CPU (DtH). Las transferencias de datos son opcionales dependiendo de las necesidades de la tarea. En la Sección 2.1 se hace un repaso al estado del arte de los modelos de ejecución concurrente de tareas en un acelerador, discutiendo algunos de los trabajos más relevantes que analizan las transferencias entre el *host* y los aceleradores, la ejecución de kernels sobre dichos aceleradores y la ejecución concurrente de comandos.

Como paso previo a la ejecución de una tarea en un acelerador se requiere la formación de dicha tarea en el *host* y, para ello, hay que reservar memoria que permita almacenar los datos que usará la tarea. En la Sección 2.3.1 de este capítulo se analizan y comparan varias alternativas para reservar la memoria.

Cada una de las etapas de una tarea se compone de uno o más comandos kernels y de uno, ninguno o varios comandos de transferencias. Los comandos enviados a la misma cola de comandos (stream en la terminología de CUDA) [1] son ejecutados en orden por lo que la concurrencia entre tareas se alcanzará entre

[1] Por simplicidad, a partir de ahora utilizaremos el término de cola de comandos para referirnos tanto a los CUDA streams como a las colas de comandos en OpenCL

comandos asíncronos lanzados en diferentes colas de comandos. En la Sección 2.2 se discute como los comandos asíncronos para transferencias y ejecución de los kernels deben ser lanzados para conseguir un solapamiento entre los mismos.

Además, para poder modelar la ejecución concurrente de tareas, es necesario estimar los tiempos de ejecución de cada tarea por separado, tanto de sus comandos de transferencia como de sus kernels. Por lo tanto, en la Sección 2.3 se presenta un modelo simple de ejecución de kernels que se usa habitualmente en la literatura, y otro de transferencias en el que hemos incluido una serie de modificaciones que nos permiten modelar las transferencias de forma más exacta.

En la Sección 2.4 se utilizan todas las aportaciones anteriores para presentar un modelo que permite simular la ejecución de un grupo de tareas y obtener una estimación del tiempo total de ejecución. A continuación, en la Sección 2.5, se presentan los benchmarks que usaremos en esta tesis para validar y comparar las aportaciones presentadas. Por último, el modelo de ejecución es validado con los benchmarks anteriores en la Sección 2.6, tanto en entornos CUDA como OpenCL.

2.1. Estado del Arte

A la hora de analizar el comportamiento de un acelerador es conveniente disponer de modelos que permitan predecir su rendimiento. Estos modelos pueden usarse después para tomar decisiones que permitan mejorar las prestaciones de un sistema heterogéneo. En los siguientes apartados se detallan algunos de los trabajos más relevantes que analizan las transferencias entre el *host* y los aceleradores, la ejecución de kernels sobre dichos aceleradores y la ejecución concurrente de comandos.

2.1.1. Modelado de Transferencias

La ejecución de uno o más kernels puede llevar consigo transferencias desde el *host* al acelerador y viceversa. Las transferencias de datos pueden tener un importante impacto en el tiempo total de ejecución de una tarea en un acelerador. Una manera de reducir tal impacto es utilizar memoria no paginable (*pinned*) en el *host*. El uso de memoria *pinned* aporta un incremento en el rendimiento de la transferencia. De esta manera, con el fin de incrementar el rendimiento, existen varios enfoques para modelar las transferencias de datos involucradas en la computación mediante aceleradores. En [11] se presenta un esquema para modelar el rendimiento en aceleradores como GPUs. En dicho trabajo se predice la canti-

dad de datos a transmitir para un conjunto de tareas, y luego se utiliza un modelo lineal de rendimiento del bus PCIe para determinar el tiempo consumido por las transferencias. En [12] se utiliza también un modelo lineal de rendimiento para predecir el tiempo de transferencia en rutinas como la multiplicación de matrices. Un modelo de transferencias más preciso es presentado en [81] donde manifiestan que en el rendimiento de una transferencia pueden intervenir varios factores como son: las arquitecturas involucradas, dirección de la transferencia y el tipo. En este trabajo los autores presentan una extensión de un modelo del bus PCI Express llamado LogGP [2, 18] cuyos parámetros pueden ser calculados mediante una simple aplicación de benchmark para medir el ancho de banda conseguido. Además, se identifican tres tipos de arquitecturas de GPUs de NVIDIA según el número de motores DMA y el tipo de sincronización realizada: (1) aceleradores con un solo motor DMA y con sincronización implícita (GTX 480 y GTX 680), (2) aceleradores con un solo motor DMA y sin sincronización implícita (GTX Titan), y (3) aceleradores con dos motores DMA y sin sincronización implícita (Tesla K20c, S2050 y GTX980). Además obtienen un modelo de rendimiento para cada categoría incluyendo el grado de solapamiento que se puede conseguir entre transferencias y computación. En [72] tienen en cuenta las transferencias de memoria con zonas de memoria no alineadas, donde se presenta una versión mejorada del modelo presentado en [81] para este tipo de transferencias.

Además de la memoria *pinned*, existen otros mecanismos para realizar transferencias de memoria entre el *host* y el acelerador, como es la memoria unificada [59]. La memoria unificada simplifica la programación, ya que permite a las aplicaciones acceder a la memoria de la CPU y la GPU sin necesidad de copiar manualmente los datos de una en otra, y facilita la introducción de soporte para aceleración en la GPU en una gran variedad de lenguajes de programación. Aunque el uso de memoria unificada simplifica la programación, hace que se pierda el control sobre las transferencias ya que el runtime se encarga de ello. En [92] se pone de manifiesto las ventajas desde el punto de vista de programación y rendimiento de ancho de banda que conlleva el uso de la memoria unificada. En este trabajo emplean una serie de mecanismos para realizar transferencias sobre la memoria unificada, los cuales podrían ser también implementados con el modelo presentado en esta tesis.

En esta tesis presentamos un modelo de transferencias similar al propuesto en [81] pero que tiene en cuenta la posibilidad de que se produzcan transferencias simultáneas en direcciones opuestas y que además permite adaptarse sobre la marcha a distintas cargas de trabajo.

2.1.2. Modelado de Kernels

En la literatura aparecen muchos esfuerzos para caracterizar el rendimiento de los kernels. Algunos modelos tienen como objetivo analizar las potenciales mejoras de rendimiento de algunas técnicas de optimización [49, 69]. En [35] se propone un modelo de predicción del rendimiento sobre CUDA basado en un conjunto de métricas del kernel y de características de la GPU. Una vez conocidas las métricas y las características de la GPU, determinan el factor limitante de rendimiento y realizan una estimación del tiempo de ejecución. Un análisis similar es desarrollado por Lemeire et al. [42] para entornos de desarrollo OpenCL. Karami et al. [33] proponen un modelo de regresión para estimar el rendimiento de los kernels en OpenCL. Este modelo necesita información de varios contadores de rendimiento de la GPU. El número de contadores (variables) que pueden influir en el rendimiento de un kernel puede ser bastante elevado, por lo que aplican un análisis de componentes principales (PCA) para reducir el número de variables en el modelo. Liu et al. [46] proponen un modelo lineal para la predicción del tiempo de ejecución de kernels en CUDA que destaca por su simplicidad y facilidad de uso.

2.1.3. Ejecución Concurrente de Comandos

La ejecución concurrente de comandos en un acelerador es soportada por distintos entornos de trabajo, como CUDA y OpenCL, por medio de un conjunto de colas software. Estas colas software se denominan Streams en CUDA [59] y colas de comandos en OpenCL [34]. Por medio de estas colas software se pueden incrementar las concurrencia de las tareas en el acelerador ya que comandos de distintas colas podrían ejecutarse simultáneamente si hay recursos hardware suficientes. Además, los aceleradores actuales presentan distintos soportes hardware que hacen posible la ejecución concurrente. Estos soportes hardware son Hyper-Q [59](NVIDIA), ACEs [4](AMD) e hilos hardware [30](Intel Xeon Phi).

La ejecución concurrente de kernels (CKE) es una característica que permite la ejecución concurrente de varios kernels o comandos de computación en el acelerador. CKE es posible cuando hay recursos computacionales disponibles en el acelerador. Algunos trabajos se han centrado en optimizar el rendimiento de aplicaciones que reparten su ejecución entre diferentes streams o colas de comandos. En [26] se obtiene el número óptimo de streams una vez conocidos los tiempos de transferencias y de kernels. Este número óptimo de streams es recalculado en tiempo de ejecución, según la carga de trabajo. En [46], demuestran que la partición de los datos y la planificación de estas particiones, tienen una gran influencia

en el rendimiento conseguido en aceleradores como NVIDIA, AMD e Intel Xeon Phi. Sin embargo, su estudio no considera el solapamiento entre transferencias realizadas en direcciones opuestas, las cuales pueden surgir en los aceleradores con dos motores DMA. Recientemente, Li et al. [43], utilizan hStreams [71] para estudiar el rendimiento obtenido con la ejecución concurrente de comandos en plataformas heterogéneas MIC como Intel Xeon Phi.

Otros trabajos se han centrado en intentar reordenar el lanzamiento de los kernels, con el objetivo de incrementar el rendimiento obtenido mediante CKE. De esta manera, Wang et al. [86] proponen una fusión de contextos para ejecutar kernels de carga ligera en aceleradores NVIDIA. Este enfoque permite que aplicaciones multihilo puedan compartir eficientemente el contexto, para poder ejecutar concurrentemente kernels. Posteriormente, Wende et al. [88] utilizan un esquema productor-consumidor para reorganizar el lanzamiento de los kernels en aceleradores con una sola cola hardware (por ejemplo en GPUs con arquitectura Fermi de NVIDIA). En este enfoque se asocia un CUDA stream a cada hilo productor (hilos CPU). Posteriormente, los diferentes CUDA streams son situados en la cola hardware para incrementar la concurrencia. En [9], Awatramani et al. utilizan GPGPUSim para mostrar como la ejecución concurrente de kernels limitados por memoria o por computación, puede incrementar la utilización de las unidades computacionales, y también reducir la carga del sistema.

Con la aparición de nuevas arquitecturas de aceleradores como NVIDIA Kepler y AMD Southern Islands, se incorporaron más colas hardware para mapear los CUDA streams (o las colas de comandos en OpenCL). El hecho de tener más colas hardware en el acelerador, evita la aparición de falsas dependencias entre kernels que son una de las consecuencias negativas de utilizar una sola cola hardware. Teniendo en cuenta esta nueva característica hardware, varios autores han desarrollado novedosas técnicas de planificación para explotar los beneficios de CKE. De este modo, Zhong et al. [91] presentan un esquema para dividir kernels de carga pesada en un conjunto de kernels de carga ligera que podrían ejecutarse concurrentemente. Además, también han desarrollado un esquema que selecciona parejas de kernels que podrían ser planificados conjuntamente. Suzuki et al. [74] presentaron un esquema que comprueba para CKE, los requisitos de memoria en el acelerador y llevan a cabo un proceso de suspensión de kernels. Todos estos trabajos se centran en explorar distintos mecanismos para explotar los beneficios de CKE, pero no tienen en cuenta, el impacto que pueden tener las transferencias de datos en el rendimiento de la aplicación.

En esta tesis presentamos un esquema para ejecutar de forma eficiente tareas concurrentes en aceleradores como GPUs NVIDIA y AMD e Intel Xeon Phi. Este esquema mejora los enfoques anteriores solapando transferencias de datos

con computación. De esta manera, podemos manejar escenarios más complejos y realistas, como escenarios multihilo donde varios hilos CPU pueden lanzar tareas al acelerador.

2.2. Lanzamiento Asíncrono de Comandos

De forma genérica, un hilo en el *host* utiliza, de forma implícita o explícita, una cola software de comandos para el envío de estos al acelerador. Esta característica se explicará con más detalle en la sección 2.4. Normalmente estos comandos son transferencias de datos entre el *host* y el acelerador, o ejecución de kernels en el acelerador. Los comandos enviados a una misma cola son ejecutados en orden por lo que no pueden solaparse entre si. Sin embargo, los comandos lanzados de forma asíncrona en diferentes colas sí pueden solaparse si existen suficientes recursos hardware. El número y tipo de comandos que se pueden solapar depende de las propiedades hardware del acelerador. De esta manera, si el acelerador tiene solamente un motor DMA, los comandos *HtD* y *DtH* no pueden solaparse entre si. Dependiendo del número de motores DMA y el mecanismo de sincronización entre comandos (con o sin sincronización implícita), se pueden identificar distintos tipos de aceleradores [81]. La sincronización implícita consiste en que un kernel lanzado por una cola de comandos, solamente puede empezar su ejecución una vez que todos los hilos de kernels lanzados anteriormente por cualquier cola de comandos hayan empezado su ejecución [59]. De forma similar, dos kernels de diferentes colas no pueden solaparse si alguno de ellos consume todos los recursos computacionales del acelerador. Por lo tanto, dependiendo del número de motores DMA y recursos computacionales disponibles, es posible solapar comandos de transferencias y de computación. Además, los aceleradores más modernos como las GPUs de NVIDIA y AMD y también los aceleradores Intel Xeon Phi, permiten la ejecución concurrente de comandos de computación de diferentes colas proporcionando varias conexiones hardware entre el *host* y el acelerador. La Tabla 2.1 muestra el número de motores DMA para los aceleradores utilizados en esta tesis.

Por otra parte, las colas software de comandos permiten asociar eventos a cada uno de los comandos. El hilo del *host* puede comprobar el estado del evento para conocer si un comando ha sido ejecutado o completado por el acelerador. De esta manera, el hilo del *host* puede usar estos eventos para introducir dependencias entre comandos lanzados en diferentes colas. El lanzamiento asíncrono de comandos y el manejo de las colas de comandos será diferente dependiendo del lenguaje de programación utilizado, así como del número de motores DMA

Acelerador	Motores DMA
NVIDIA K20c	2
NVIDIA GTX 980	2
NVIDIA Titan X	2
Intel Xeon Phi	1
AMD R9	2

Tabla 2.1: Número de motores de DMA para los aceleradores NVIDIA K20c, GTX 980 y Titan X, Intel Xeon Phi y AMD R9. Estos aceleradores son los utilizados en esta tesis.

del acelerador. De esta manera, en esta tesis proponemos distintos esquemas de lanzamiento para CUDA y OpenCL, distinguiendo entre arquitecturas con uno o dos motores DMA.

2.2.1. CUDA Streams

Para los programadores CUDA las colas de comandos se denominan streams. Los comandos lanzados en un mismo stream son ejecutados en orden y no pueden solaparse. Sin embargo, comandos lanzados en diferentes streams pueden ejecutarse fuera de orden y de forma concurrente. Estos streams son asignados a colas hardware en el acelerador, de forma que comandos de diferentes colas puedan ser ejecutados concurrentemente. Por lo tanto, los hilos del *host* envían asíncronamente comandos a streams y el acelerador planifica comandos de diferentes streams cuando hay recursos disponibles. Más precisamente, los comandos de transferencia de memoria son ejecutados por los motores DMA y los comandos de computación se ejecutan por los elementos de computación del acelerador. En esta tesis hemos centrado nuestro estudio en los arquitecturas de NVIDIA más habituales hoy en día, las cuales disponen de dos motores DMA como por ejemplo las GPUs K20, GTX980 y Titan X.

La Figura 2.1 muestra la ejecución de tres tareas en un acelerador con dos motores DMA. Por simplicidad, asumimos que los kernels consumen todos los recursos computacionales por lo que la ejecución concurrente de kernels no tiene lugar. Como recomienda la guía de programación de CUDA [59], los comandos pertenecientes a una misma tarea son lanzados por el mismo stream. El hilo del *host* lanza los comandos agrupándolos por tareas, para aumentar el tiempo en que los dos motores DMA están trabajando al mismo tiempo. De esta manera, cuando los comandos *HtD*, *K* y *DtH* de la tarea 0 son enviados (indicados como HS_0, KS_0 y DS_0), estos son lanzados en el stream 0. La propia sincronización impuesta por el stream a los comandos que contiene mantiene las dependencias internas de

la tarea. Se puede observar que tras ejecutarse HTD_0 empiezan a ejecutarse de forma concurrente K_0 y HTD_1, y que la máxima concurrencia ocurre cuando se están ejecutando al mismo tiempo DTH_0, K_1 y HTD_2.

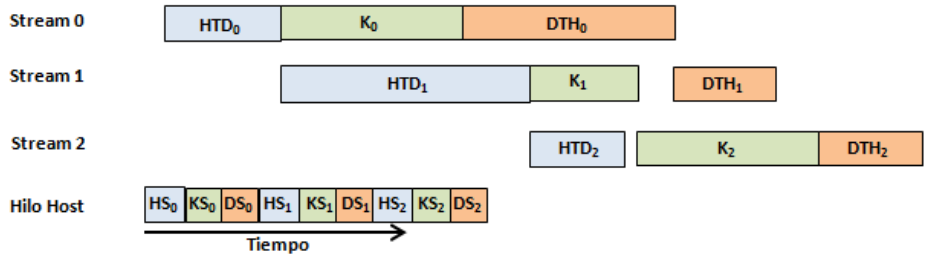
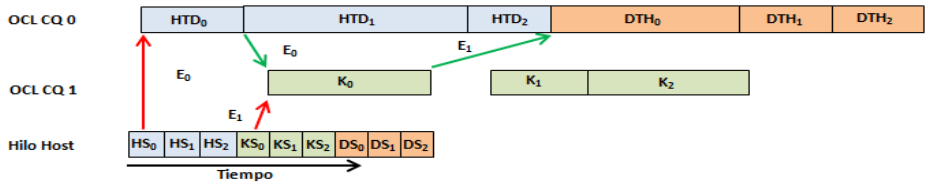


Figura 2.1: Esquema de lanzamiento para CUDA y aceleradores con dos motores DMA. Este esquema utiliza un stream para cada tarea. El hilo del *host* lanza los comandos agrupándolos por tareas. Las dependencias internas de una tarea son mantenidas por medio de la sincronización implícita impuesta por el stream en los comandos que son lanzados dentro de él.

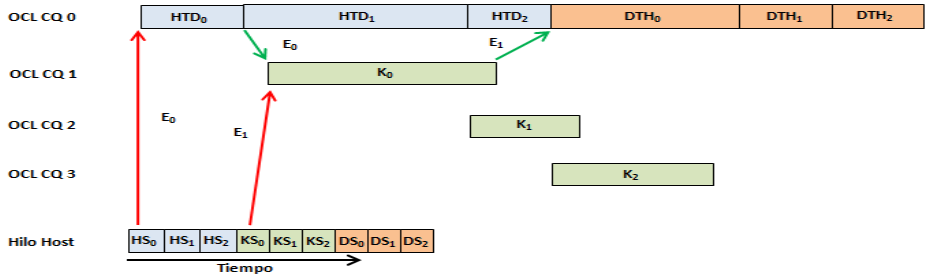
2.2.2. Colas de Comandos en OpenCL

Para entornos de programación OpenCL, la manera en que los comandos de una tarea deben ser enviados a las colas dependen de la asignación de estas a las colas hardware del acelerador. Teniendo en cuenta las características específicas del acelerador y algunas restricciones para mejorar el rendimiento de las tareas, en esta tesis proponemos dos esquemas de lanzamiento basados en el número de motores de DMA del acelerador y la decisión de habilitar CKE. Los esquemas de lanzamiento propuestos en esta sección para OpenCL y aceleradores con un motor DMA son igualmente válidos para entornos CUDA. Cuando tratamos con aceleradores con dos motores DMA, hay que tener en cuenta como OpenCL asigna sus colas hardware a los motores DMA. Concretamente, OpenCL asocia las colas pares e impares a motores DMA diferentes, por lo que los comandos de transferencia según su tipo deben ser lanzados por colas pares o impares [4] para conseguir que las transferencias en sentidos opuestos se solapen.

En la Figuras 2.2(a) y 2.2(b) mostramos los esquemas propuestos en OpenCL para lanzar asíncronamente tareas en aceleradores con un solo motor DMA (por ejemplo Intel Xeon Phi 5100 series). Al igual que en ejemplos anteriores, en estos esquemas se lanzan tres tareas asíncronamente. La Figura 2.2(a) muestra el esquema para un motor DMA sin habilitar CKE. El hilo del *host* lanza los



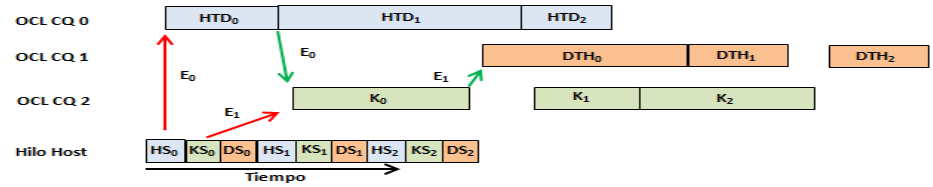
(a) Esquema de lanzamiento para OpenCL sin habilitar CKE en aceleradores con un solo motor de DMA. Este esquema usa dos CQs para lanzar asíncronamente tres tareas. Una CQ se usa para las transferencias de memoria y otra para computación.



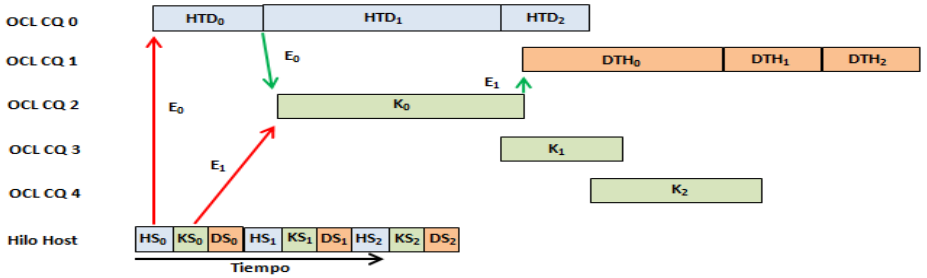
(b) Esquema de lanzamiento para OpenCL habilitando CKE en aceleradores con un solo motor DMA. Este esquema usa cuatro CQs para lanzar asíncronamente tres tareas. Una CQ se usa para las transferencias de memoria y tres CQs diferentes (una por cada tarea o kernel) para computación.

Figura 2.2: Esquemas de lanzamiento en OpenCL para aceleradores con un motor DMA y sin sincronización implícita. El número de colas empleadas en estos esquemas puede variar dependiendo de si se habilita CKE o no. En ambos esquemas, el hilo del *host* lanza los comandos agrupándolos por tipo. Además, las dependencias internas de una tarea tienen que ser manejadas por el hilo del *host* a través de eventos OpenCL.

comandos agrupándolos por tipo, por un lado todas las transferencias se lanzan por la cola de comandos CQ 0, y por otro lado la ejecución de kernels por la cola CQ 1. Al ser lanzados todos los comandos de kernels por la misma cola, no hay posibilidad de que se puede llevar a cabo la ejecución concurrente de kernels. El orden de inserción en las colas es primero todos los comandos *HtD*, luego los comandos *K* y finalmente los comandos *DtH*. Como todos los comandos *HtD* son enviados antes que los comandos *DtH*, se reduce el tiempo de inactividad que podría haber en el motor DMA por las dependencias entre los comandos *K* y los comandos *DtH*. Los comandos *K* son enviados a la cola CQ 1 para conseguir el solapamiento con los comandos de transferencias. Para permitir la ejecución



(a) Esquema de lanzamiento para OpenCL, aceleradores con dos motores DMA sin habilitar CKE. Este esquema utiliza tres CQs para lanzar asíncronamente tres tareas. Dos CQs se emplean para lanzar los comandos de transferencias y otra CQ para los comandos de computación.



(b) Esquema de lanzamiento para OpenCL, aceleradores con dos motores DMA y habilitando CKE. Este esquema utiliza cinco CQs para lanzar asíncronamente tres tareas. Dos CQs se emplean para lanzar los comandos de transferencias y tres CQs diferentes (una por cada tarea o kernel) para los comandos de computación.

Figura 2.3: Esquemas de lanzamiento para OpenCL y aceleradores con dos motores DMA y sin sincronización implícita. El número de colas empleadas en estos esquemas puede variar dependiendo de si se habilita CKE. OpenCL asocia las colas pares e impares a motores DMA diferentes, por lo tanto los comandos *HtD* y *DtH* se lanzan en las CQs 0 y 1 respectivamente. El hilo del *host* lanza los comandos agrupándolos por tareas. Además, este se encarga también de mantener las dependencias internas de una tarea mediante eventos OpenCL asociados a los comandos.

concurrente entre kernels, se pueden utilizar varias colas para lanzar los comandos K como se muestra en la Figura 2.2(b). Para garantizar la ejecución correcta de las tareas, cuando un comando *HtD* o K es enviado por un hilo del *host* (indicado como HS_0 y KS_0), se asocia un evento OpenCL al comando (indicado como E_0 y E_1 en las Figuras 2.2(a) y 2.2(b)). Este evento cambia su estado de enviado a completado cuando su ejecución ha terminado. Las flechas rojas y verdes en la Figuras 2.2(a) y 2.2(b) indican el instante de tiempo en que el evento es enviado y completado, respectivamente. Por lo tanto, la ejecución del comando K de la tarea 0 es retrasada hasta que el evento E_0 ha alcanzado su estado de completado.

De formar similar, el comando *DtH* de la tarea 0 no empezará su ejecución hasta que el evento E_1 haya alcanzado el estado de completado.

Para aceleradores con dos motores DMA, como por ejemplo AMD R9 o NVIDIA K20c, los esquemas de lanzamiento propuestos se muestran en las Figuras 2.3(a) y 2.3(b). En estos esquemas se emplean dos colas para lanzar los comandos *HtD* y *DtH*, ya que las transferencias en direcciones opuestas pueden llevarse a cabo de forma concurrente. La cola asociada a cada comando de transferencia es importante porque OpenCL asocia las colas pares e impares a motores DMA diferentes [4]. Por lo tanto, los comandos *HtD* y *DtH* son lanzados en las colas CQ 0 y CQ 1 respectivamente. La Figura 2.3(a) muestra el esquema en OpenCL para dos motores DMA sin habilitar CKE. En este esquema se emplea solamente una cola, CQ 2, para enviar los comandos de computación *K*, pero al igual que en los esquemas anteriores podrían usarse varias colas para permitir la ejecución concurrente entre kernels, como se muestra en la Figura 2.3(b). Por otra parte, a diferencia de los esquemas para un solo motor DMA, el hilo del *host* inserta los comandos en sus colas correspondientes agrupados por tarea. De esta manera, se incrementa el tiempo en que los dos motores DMA están trabajando al mismo tiempo. Debido a que los comandos pertenecientes a una tarea son lanzados por colas distintas, el hilo del *host* debe encargarse de mantener las dependencias internas de las tareas. Estas dependencias son fijadas mediante la asociación de eventos OpenCL a los comandos, igual que en el esquema anterior.

2.3. Estimación de Tiempos de Ejecución de los Comandos

El modelo de ejecución de grupos de tareas que presentaremos en los capítulos siguientes necesita de una estimación previa de los tiempos de ejecución de los distintos comandos. El tiempo de ejecución de los comandos de transferencias de memoria puede variar dependiendo del tipo de memoria reservada para los datos. En esta sección explicaremos las diferentes formas de reservar memoria en el *host* y como influyen cada una de ellas en los tiempos de ejecución de los comandos de transferencias. Además, explicaremos como se lleva a cabo la estimación de los comandos que constituyen una tarea.

2.3.1. Reserva y Liberación de Memoria

Las transferencias de datos entre CPU y el acelerador requieren de una reserva de memoria en CPU. Tanto la guía de programación de CUDA [59] como de OpenCL [34] recomiendan el uso de memoria no paginable para explotar al máximo el ancho de banda del bus PCIe. La memoria no paginable es un tipo de memoria que reside en la memoria DRAM del *host* y no puede ser trasladada a memoria de disco. Una GPU siempre tiene que realizar transferencias desde y hacia memoria no paginable. Por lo tanto, si el driver del acelerador tiene que hacer una transferencia, primero debe realizar una copia de datos desde memoria paginable hacia memoria no paginable, y posteriormente realizar la transferencia DMA. Por el contrario, si el programador reserva directamente memoria no paginable, la transferencia DMA puede ser realizada directamente por el driver hacia la GPU o acelerador. El uso de memoria no paginable conlleva varios beneficios como la ejecución concurrente de transferencias de memoria con la ejecución de kernels, así como el mapeo directo de este tipo de memoria en el espacio de direcciones del acelerador. Sin embargo, este tipo de memoria es un recurso escaso por lo que si se hace un uso intensivo de ella puede llevar a una degradación del rendimiento del sistema. Cuando tratamos con OpenCL, en Margiolas et al. [48] se afirma que la reserva de este tipo de memoria puede, por un lado, mejorar el rendimiento de la comunicación, pero por otro, consumir demasiado tiempo en el proceso de reserva. Para obtener el mejor compromiso entre tiempos de transferencia y reserva de memoria proponen cuatro políticas:

- *Standard*: utiliza funciones de reserva de memoria de lenguaje C (por ejemplo: *malloc*).
- *OpenCL*: utiliza funciones de reserva de memoria propias de la librería OpenCL.
- *Standard with Locking*: emplea funciones de memoria pertenecientes a la librería POSIX (por ejemplo: *mlock*).
- *Hybrid*: combina funciones de memoria de las librerías de OpenCL y POSIX.

En esta tesis hemos elegido la política de *OpenCL* para obtener el mejor rendimiento. Esta política reserva un segmento de memoria en CPU y luego es añadido al espacio de memoria de la aplicación mediante la función *clEnqueueMapBuffer*. Las Figuras 2.4.a y 2.4.b muestran el rendimiento conseguido por transferencias *HtD* y *DtH* usando memoria no paginable y paginable. La Figura 2.4.a muestra, para entornos CUDA, los tiempos de ejecución alcanzados en transferencias

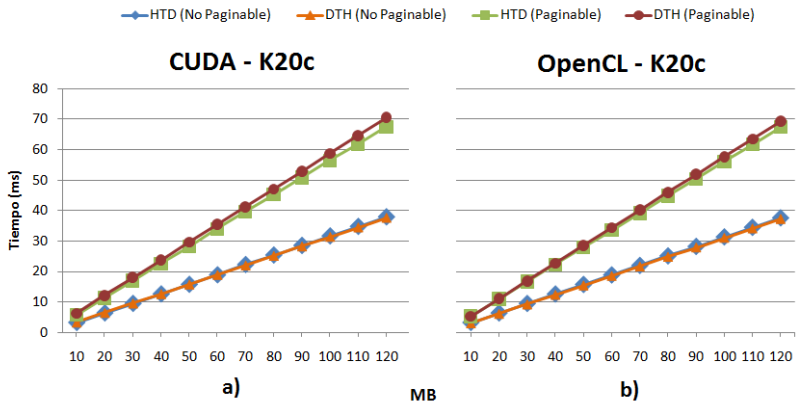


Figura 2.4: Rendimiento alcanzado por las transferencias de memoria cuando se usa memoria paginable y memoria no paginable. Las transferencias han sido realizadas para CUDA y OpenCL, desde 10 MB hasta 120 MB, sobre un bus PCIe 2.0 y para el acelerador NVIDIA K20c.

HtD y *DtH*, con memoria paginable y no paginable. Las transferencias han sido ejecutadas con tamaños que varían desde 10 MB hasta 120 MB, sobre un bus PCIe 2.0 y para el acelerador NVIDIA K20c. Esta misma situación también ha sido reproducida para entornos OpenCL, como se muestra en la Figura 2.4.b. En ambas figuras se puede observar que el uso de memoria no paginable hace que se explote de una manera más eficiente el ancho de banda del bus PCIe.

2.3.2. Transferencias de Memoria

El tiempo consumido por las transferencias *HtD* y *DtH* se puede estimar con un modelo del bus PCI Express, como el presentado por Werkhoven et al. [81]. En ese trabajo los autores presentan una extensión de un modelo del bus PCI Express llamado LogGP [2, 18], cuyos parámetros pueden ser medidos mediante una simple aplicación de benchmark. La estimación de dos transferencias en direcciones opuestas también es considerada por este modelo. El modelo funciona correctamente cuando no existe solapamiento o existe un solapamiento completo de las transferencias, pero no funciona cuando el solapamiento entre ellas ocurre de una forma parcial. Por tanto, hemos desarrollado un modelo más preciso para cualquier grado de solapamiento entre las transferencias. Nuestro modelo intro-

duce un nuevo parámetro, λ, que varía entro 0 y 1 para modelar el solapamiento de un comando de transferencia con otra transferencia en sentido opuesto. Es decir, supongamos que G^1 es el tiempo por byte alcanzado cuando la transferencia tiene lugar sola, G^2 el tiempo por byte conseguido cuando dos transferencias se realizan concurrentemente, y λ la fracción de bytes transferidos concurrentemente. Partiendo del modelo LogGP se puede expresar el tiempo de transferencia t_t de los mensajes de tamaño k bytes mediante la siguiente ecuación:

$$t_t = L + o + (1 - \lambda)kG^1 + \lambda kG^2 \tag{2.1}$$

donde L es la latencia ocasionada por el envío de un mensaje desde un punto a otro punto, y o es el tiempo que la CPU consume en registrar la petición de transferencia en el motor DMA mediante el controlador. La Ecuación 2.1 puede predecir con más exactitud el tiempo de transferencia para cualquier grado de solapamiento, en contraposición a modelos más simples que solamente consideran cuando no hay solapamiento o cuando existe un solapamiento completo de las transferencias.

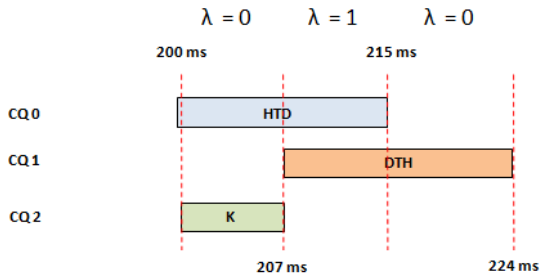


Figura 2.5: Ejemplo de simulación del solapamiento de transferencias. El modelo utiliza un nuevo parámetro λ que varía entre 0 y 1 para simular el grado de solapamiento entre las transferencias.

La Figura 2.5 muestra un ejemplo donde se detecta el solapamiento entre dos transferencias. Para esta figura, supongamos que el comando *HtD* tiene que transmitir x bytes, el comando DTH tiene que transmitir y bytes y nos encontramos en el instante de tiempo $T_a = 200$ ms. A priori no es posible aplicar la Ecuación 2.1 ya que no se sabe cuál será el valor de λ para cada transferencia, pero sí es posible ajustar su valor por tramos considerando por separado las partes con solapamiento ($\lambda = 1$) de las partes sin solapar ($\lambda = 0$), tal y como se explica a continuación. En el instante de tiempo T_a llega el comando *HtD* y como

solamente se van a ejecutar los comandos *HtD* y *K*, la predicción del tiempo de ejecución de *HtD* usando la Ecuación 2.1 tomaría los valores $\lambda = 0$ y $k = x$, quedando de la forma $t_{HtD} = T_a + L + o + xG^1$ siendo t_{HtD} la estimación actual del final del comando *HtD*. A continuación, en el instante de tiempo $T_b = 207$ ms, el comando *K* ha acabado y entra un nuevo comando *DtH*. En este momento se calculan los bytes que han sido transmitidos por *HtD* hasta ese instante T_b. Para ello utilizamos de nuevo la Ecuación 2.1, para despejar la variable k, quedando la fórmula $k = (L + o + t_t)/G^1$, donde $t_t = 7$ es el tiempo transcurrido para la transferencia *HtD*. De esta forma, se calcula la cantidad de bytes transmitidos por la transferencia *HtD* cuando no existe ningún solapamiento con otra transferencia. Por tanto, los bytes restantes por transmitir para la transferencia *HtD* serán $x - k$. En este mismo instante de tiempo ha empezado a ejecutarse la transferencia *DtH* concurrentemente con *HtD*, por lo que se deben recalcular los tiempos de finalización de las transferencias con $\lambda = 1$. Es decir, para la transferencia *HtD* la Ecuación 2.1 quedaría de la forma $t_{HtD} = T_b + L + o + (x - k)G^2$ y para *DtH* de la forma $t_{DtH} = T_b + L + o + yG^2$. El siguiente evento ocurre al finalizar la transferencia *HtD*, en $T_c = 215$ ms. A partir de este instante de tiempo solamente se ejecuta la transferencia *DtH*, por lo que se debe calcular para esta transferencia los bytes transmitidos hasta el momento. Para ello, utilizamos de nuevo la Ecuación 2.1 para despejar la variable k, quedando la fórmula $k = (L + o + t_t)/G^2$, donde $t_t = 9$ es el tiempo transcurrido para la transferencia *DtH*. Por último, se recalcularía el tiempo de finalización de *DtH* con $\lambda = 0$ (no hay solapamiento de transferencias) de la forma $t_{DtH} = T_c + L + o + (y - k)G^1$.

La Figura 2.6 muestra el error relativo en valor absoluto del tiempo predicho por nuestro modelo, denominado modelo de solapamiento parcial, en CUDA (NVIDIA K20c) y OpenCL (AMD R9). El error cometido por nuestro modelo ha sido comparado con el error cometido por los modelos con solapamiento y sin solapamiento. En este experimento se han usado transferencias asíncronas y memoria no paginable en el *host*. Una cola de comandos se usa para lanzar una transferencia asíncrona *HtD*, mientras que en otra cola de comandos se lanza una transferencia asíncrona *DtH*, que solapa un 0 %, 25 %, 50 %, 75 % y 100 % con el otro comando. El experimento ha sido llevado a cabo usando diferentes tamaños de transferencias, entre 16 MB y 512 MB, para evaluar la precisión de la predicción de los modelos. Una vez obtenida la predicción del tiempo de los comandos, se mide el tiempo real de ejecución de estos y se calcula el error de la predicción para los tres modelos. La Figura 2.6 muestra como la predicción de nuestro modelo es mejor para cada uno de los grados de solapamiento. La Tabla 2.2 muestra (resumiendo los datos de la Figura 2.6) los porcentajes máximo, mínimo y medio del error relativo en valor absoluto, de la predicción para

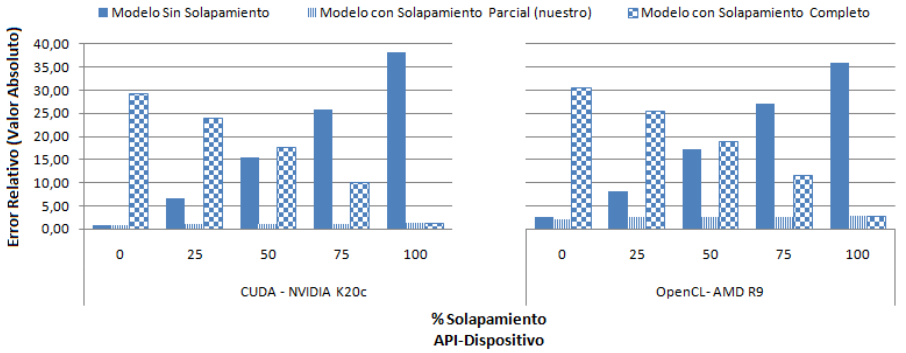


Figura 2.6: Error relativo en valor absoluto de la predicción para transferencias bidireccionales con distintos grados de solapamiento. La figura muestra los valores del error cometido para CUDA (NVIDIA K20c) y OpenCL (AMD R9). En la figura se consideran tres modelos: modelo sin solapamiento, modelo con solapamiento parcial y modelo con solapamiento completo.

API/Dispositivo	CUDA NVIDIA K20c			OpenCL AMD R9		
% Error Relativo	Max.	Min.	Media	Max.	Min.	Media
Sin Solapamiento	38.25	0.91	17.42	35.82	2.59	18.18
Con Solapamiento Parcial (Nuestro)	1.33	0.91	1.11	2.84	2.23	2.47
Con Solapamiento Completo	29.35	1.33	16.45	30.46	2.75	17.89

Tabla 2.2: Porcentajes máximo, mínimo y medio del error relativo de predicción para todos los grados de solapamiento de transferencias bidireccionales usando tres modelos: modelo sin solapamiento, modelo con solapamiento parcial y modelo con solapamiento completo.

todos los grados de solapamiento usando los tres modelos. En esta tabla se puede observar que el porcentaje medio del error relativo en valor absoluto para nuestro modelo está por debajo de 1.2 % para CUDA (NVIDIA K20c) y de 2.6 % para OpenCL (AMD R9).

2.3.3. Kernels

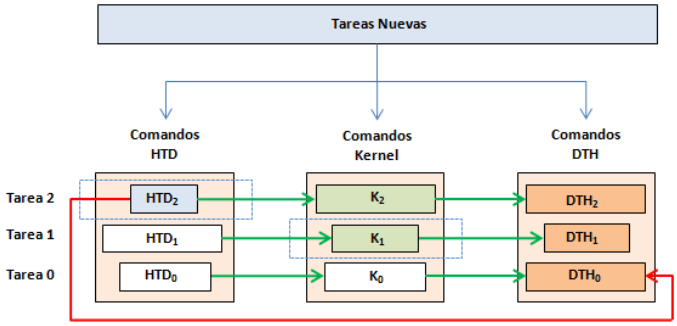
Hay muchos esfuerzos en la literatura para caracterizar el rendimiento de los kernels ejecutados en el acelerador. Algunos modelos complejos tienen como objetivo analizar las potenciales mejoras de rendimiento de algunas técnicas de optimización [49, 69], pero el objetivo de este trabajo no es mejorar el código del kernel sino predecir su tiempo de ejecución. Por ello, hemos empleado un modelo lineal simple como el usado por Liu et al. [46]. En ese trabajo, para un kernel dado y un número de elementos de datos de entrada igual a m, el tiempo de computación es estimado mediante la siguiente ecuación:

$$T = \eta \cdot m + \gamma \tag{2.2}$$

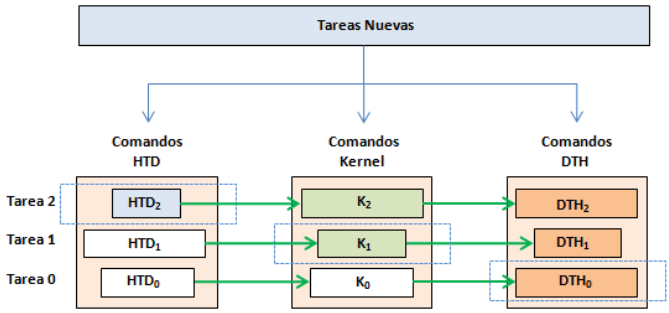
donde η es la tasa de computación definida a partir del tiempo de ejecución por dato de entrada, y γ es la latencia de invocación del kernel. Nuestro modelo de ejecución de tareas concurrentes solamente necesita mantener un registro de estos dos parámetros, que puede ser obtenido a partir de ejecuciones previas de los kernels que se quieran planificar. También es posible obtener el tiempo de ejecución de los kernels a partir de ejecuciones iniciales tal y como hacen en OmpSs o StarPU [8, 21].

2.4. Modelo General. Colas de Simulación

Tal y como se ha indicado previamente, una tarea está compuesta por tres etapas *HtD-K-DtH* que deben ejecutarse en orden. Las etapas *HtD* y *DtH* pueden estar compuestas por uno, ninguno o varios comandos y la etapa K por el contrario, puede estar compuesta por uno o varios comandos. Teniendo en cuenta los esquemas de lanzamiento presentados en la Sección 2.2, tanto para CUDA como para OpenCL, proponemos un modelo que utiliza tres colas software FIFO para simular la computación de un grupo de tareas (*Task Group*, *TG* a partir de ahora). Cada cola es responsable de la simulación de un tipo de comandos. Estas colas software no son independientes, ya que existen dependencias explícitas entre los comandos pertenecientes a una misma tarea y dependencias no explícitas debidas a las restricciones hardware del acelerador. Las Figuras 2.7(a) y 2.7(b) representan las dependencias entre las diferentes colas para aceleradores con uno y dos motores DMA respectivamente. La cabeza de cada cola ha sido marcada mediante un rectángulo discontinuo azul. Por lo tanto, el estado de simulación mostrado en la Figura 2.7(a) indica que los comandos HtD_2 y K_1 están siendo



(a) Modelo para aceleradores con un solo motor DMA.



(b) Modelo para aceleradores con dos motores DMA.

Figura 2.7: Las flechas verdes entre comandos de diferentes colas, representan las dependencias internas de una tarea. Las flechas rojas simulan la dependencia cuando se envían comandos HtD y DtH en aceleradores con un motor DMA (los comandos DtH son enviados justo después de que todos los comandos HtD han sido enviados). Los comandos son insertados en sus respectivas colas y lanzados en ese orden. Los comandos que están actualmente en simulación son marcados con rectángulos azules con líneas discontinuas. Por el contrario, los comandos que han sido simulados son representados mediante cajas de color blanco.

simulados. De forma similar, las cajas blancas indican los comandos que ya han sido simulados, mientras que los comandos restantes están esperando a que se resuelvan sus dependencias explícitas (flechas verdes) y no explícitas (flechas rojas). De esta manera, en la Figura 2.7(a) se muestra que el comando DtH_0 no puede ser simulado aún debido a que no se ha resuelto su dependencia no explícita con el comando HtD_2. Esta dependencia simula el comportamiento del esquema propuesto para OpenCL y aceleradores con un solo motor DMA. La Figura 2.7(b) representa la misma situación descrita anteriormente pero para aceleradores con

dos motores DMA. En la Figura 2.7(b) se puede apreciar que el comando DtH_0 no mantiene una dependencia explícita con el comando HtD_2 debido a que existen dos motores DMA. Por tanto, el comando DtH_0 puede ser simulado cuando se resuelve su dependencia implícita con el comando K_0.

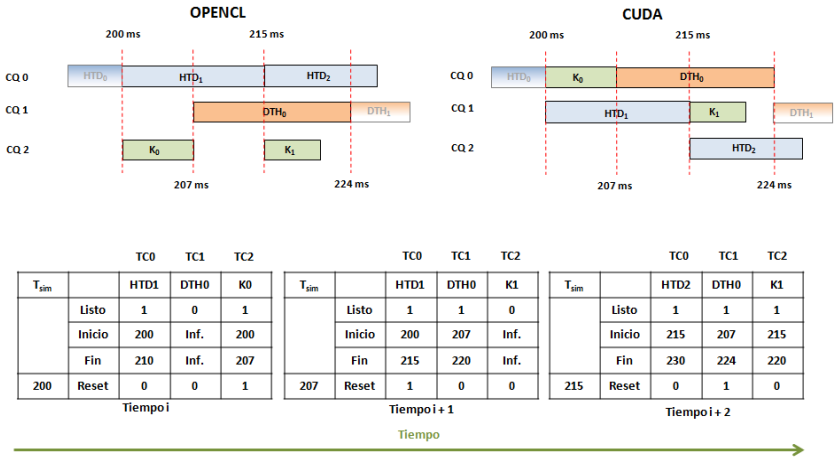


Figura 2.8: Ejemplos de la simulación para OpenCL y CUDA, de la ejecución de varios comandos pertenecientes a tres tareas diferentes. Las líneas discontinuas rojas verticales identifican los pasos de la simulación. La información calculada en estos tres pasos de simulación se muestra en las tablas de abajo, que son válidas para entornos OpenCL y CUDA. En cada paso de la simulación, los comandos listos en la cabeza de las colas FIFO son identificados y se calculan sus tiempos de ejecución de inicio y fin.

La Figura 2.8 muestra un ejemplo, válido para entornos OpenCL y CUDA, de como el simulador calcula el tiempo de ejecución de un *TG*. El acelerador simulado en el ejemplo tiene dos motores DMA. Para ambos entornos se utilizan tres colas para el envío de los comandos (ver Figura 2.1 para el caso de CUDA y Figura 2.3(a) para el caso de OpenCL). Independientemente del entorno de programación, se crean tres estructuras de información (*TC0, TC1* y *TC2*), una por cada tipo de comando (*HtD*, *DtH* y *K*), donde se anotan el tiempo de inicio y fin de los comandos en cada paso de la simulación. Estas estructuras también tiene dos flags que toman valores 0 o 1 para indicar si el comando está listo (Listo) o si debe resetearse la información de la estructura en el siguiente paso de simulación (Reset). Los ejemplos comienzan en el tiempo de simulación $T_{sim} = 200$ ms (Tiempo i en la Figura 2.8), donde HtD_0 acaba de finalizar y ha sido borrado de la cabeza de la cola FIFO asociada a los comandos *HtD* ($FIFO_{HtD}$). En este

instante de tiempo, dos comandos están listos en las cabezas de sus colas FIFO: HtD_1 y K_0 (Listo = 1) localizados en las colas $FIFO_{HtD}$ y $FIFO_K$ respectivamente. A continuación, se estiman los tiempos de inicio y fin de los comandos listos y son anotados en las estructuras *TC0* y *TC2*. La estimación de los tiempos de ejecución de los comandos se basa en los modelos de ejecución previamente presentados en la Sección 2.3. Después de esto, el instante de simulación avanza hasta el tiempo de fin más próximo de los comandos listos. Por lo tanto, en este ejemplo el instante de simulación es actualizado a $T_{sim} = 207$ ms, cuando finaliza el comando K_0 (Reset = 1). Este instante de simulación es el Tiempo i+1 en la Figura 2.8. Ahora K_0 es borrado de la cola $FIFO_K$ y DtH_0 pasa a estar listo (Listo = 1), ya que su dependencia con el comando K_0 ha sido resuelta. A continuación, se anotan los tiempos de inicio y fin de los comandos listos. Sin embargo, se detecta un solapamiento de transferencias entre los comandos HtD_1 y DtH_0, por lo tanto se calcula el grado de solapamiento entre los dos comandos, se recalculan sus tiempos de fin y se anotan de nuevo en las estructuras *TC0* y *TC1* (ver la Sección 2.3.2 para más detalles). Hay que indicar que, después de detectar este solapamiento entre comandos de transferencias, el tiempo de finalización de HtD_1 ha cambiado de 210 a 215 ms. Una vez que los tiempos de finalización han sido calculados correctamente, el simulador avanza su instante de simulación hasta el tiempo de finalización más próximo de los comandos listos, es decir, hasta $T_{sim} = 215$ (Tiempo i+2 en la Figura 2.8). En este instante de simulación, HtD_1 es desencolado de la cola $FIFO_{HtD}$ y tanto HtD_2 como K_1 pasan a estar listos. El simulador detecta un nuevo solapamiento de transferencias entre los comandos DtH_0 y HtD_2 y, consecuentemente, se realiza una nueva estimación de tiempos de finalización de ambos comandos. Como consecuencia, el tiempo de finalización del comando DtH_0 cambia de 220 a 224 ms.

Como se puede observar en el ejemplo anterior, nuestro modelo evita la ejecución concurrente de kernels (CKE), ya que emplea una única cola software para la simulación de los comandos *K*. Hay dos razones para esto. Por una parte, la mejora obtenida por CKE cuando los kernels agotan uno o varios recursos del acelerador (registros, memoria compartida o caché, número de hilos lanzados, etc.) es muy limitada, y son precisamente este tipo de kernels los que más frecuentemente se ejecutan en los aceleradores. En esta situación, Hyper-Q (NVIDIA) solamente puede solapar una pequeña parte de un kernel. Esta pequeña parte corresponde con el final de un kernel (solo cuando algunos recursos están siendo liberados) y el comienzo del otro kernel concurrente. La mejora obtenida por este pequeño solapamiento no justifica el incremento de complejidad del modelo cuando se requieren más de una cola de comandos para el lanzamiento de los kernels (una cola de comandos para cada kernel concurrente). Por otra parte,

el mecanismo empleado por los aceleradores para implementar CKE no ha sido hasta ahora revelado por ningún fabricante. Esto hace muy difícil desarrollar un modelo genérico para la ejecución concurrente de kernels, el cual es requerido por nuestro enfoque para la estimación del tiempo de ejecución de estos comandos. Aún así, en nuestros resultados experimentales mostraremos que el orden predicho por nuestro modelo es capaz de mejorar la mayoría de ordenes de ejecución cuando se utiliza CKE.

2.5. Benchmarks

Para la comprobación tanto del modelo de ejecución de tareas como del método de selección del orden óptimo de ejecución, que se presentarán en los próximos capítulos, hemos definido un conjunto de tareas sintéticas y reales que presentan tiempos de transferencias de datos y de computación distintos para conseguir unas cargas de trabajo lo más representativas y variadas posibles.

2.5.1. Sintéticos

El conjunto de tareas sintéticas se obtiene a partir de la ejecución de unos códigos que permiten variar los tiempos de ejecución y transferencia. Los Códigos 2.1 y 2.2 muestran la función que es ejecutada por estas tareas en CUDA y OpenCL respectivamente. El parámetro *input*, junto con una fracción f que varía entre 0.05 y 0.9, se puede usar para fijar la duración de los comandos *HtD* y *DtH*. Este parámetro es un puntero a un vector de k bytes reservado en la memoria del acelerador. El valor de k se selecciona usando la Ecuación 2.1, con $\lambda = 0$, para conseguir una transferencia de 10 ms. La fracción f es usada para limitar la transferencia a $f \cdot k$ bytes, de esta forma el tiempo de transferencia t_t varía en nuestros experimentos entre 0.5 ms y 9 ms, aproximadamente. Los Códigos 2.3 y 2.4 muestran los métodos para lanzar los comandos *HtD* y *DtH* para una tarea sintética en CUDA y OpenCL respectivamente. En estos códigos se puede observar como se utiliza la fracción f (f_htd o f_dth) para ajustar el tiempo deseado de la transferencia. Esta fracción es multiplicada por una cantidad de bytes prefijada de antemano ($elements\_htd * sizeof(int)$ o $elements\_dth * sizeof(int)$) cuyo tiempo de transferencia tiene que ser 10 ms. De forma similar, diferentes valores de la variable *num_iteraciones* conducen a tiempos de computación del kernel diferentes y se puede ajustar para que los tiempos oscilen también entre 0.5 ms y 9 ms. La variable *num_iteraciones* es el resultado de multiplicar una fracción f por un número de iteraciones prefijado de antemano para que el tiempo de

ejecución del kernel sea igual a 10 ms.

```
__global__ void kernel_sintetico(int *input, int num_iteraciones,
    int factor)
{
int idx = blockDim.x * blockIdx.x + threadIdx.x;

for(int i=0;i<num_iteraciones;i++)
    input[idx] *= factor;
}
```

Código 2.1: Código del kernel CUDA para una tarea sintética.

```
__kernel void kernel_sintetico(__global int *input, int
    num_iteraciones, int factor)
{
int idx = get_global_id(0);

for(int i=0;i<num_iteraciones;i++)
    input[idx] *= factor;
}
```

Código 2.2: Código del kernel OpenCL para una tarea sintética.

```
//Transferencia HtD
cudaMemcpyAsync(input, host_input, f_htd*elements_htd*sizeof(int),
cudaMemcpyHostToDevice, stream);

//Transferencia DtH
cudaMemcpyAsync(host_input, input, f_dth*elements_dth*sizeof(int),
cudaMemcpyDeviceToHost, stream);
```

Código 2.3: Código CUDA para las transferencias HtD y DtH una tarea sintética.

```
//Transferencia HtD
clEnqueueWriteBuffer(*cq, input, CL_FALSE, 0, f_htd*elements_htd*
    sizeof(int), host_input, 0, NULL, NULL);

//Transferencia DtH
clEnqueueReadBuffer(*cq, input, CL_FALSE, 0,
                    f_dth*elements_dth * sizeof(int),
                    host_input, 0, NULL, NULL);
```

Código 2.4: Código OpenCL para las transferencias HtD y DtH una tarea sintética.

Este formato de especificación de la duración de los comandos permite aumentar o disminuir fácilmente los tiempos, de forma que si fuera necesario es posible considerar comandos que, por ejemplo, duren 1 s en lugar de 1 ms sin cambiar nada del código. Además, esta forma de definir las transferencias y la computación

Tarea Sintética	T 0	T 1	T 2	T 3	T 4	T 5	T 6	T 7	T 8	T 9	T 10	T 11
HtD	0.1	0.2	0.3	0.1	0.05	0.1	0.25	0.05	0.05	0.05	0.15	0.2
K	0.8	0.7	0.6	0.7	0.4	0.6	0.5	0.9	0.8	0.8	0.7	0.6
DtH	0.1	0.1	0.1	0.2	0.05	0.3	0.25	0.05	0.05	0.15	0.15	0.2

Tabla 2.3: Tareas sintéticas de kernel dominante usadas en nuestros benchmarks. Los comandos *HtD*, *K* y *DtH* se definen con una fración de tiempo con respecto a una unidad de tiempo. La unidad de tiempo es 10 ms.

Tarea Sintética	T 12	T 13	T 14	T 15	T 16	T 17	T 18	T 19	T 20	T 21	T 22	T 23
HtD	0.3	0.4	0.4	0.4	0.5	0.5	0.5	0.6	0.6	0.7	0.8	0.9
K	0.4	0.4	0.3	0.2	0.4	0.3	0.2	0.3	0.2	0.2	0.1	0.05
DtH	0.3	0.2	0.3	0.4	0.1	0.2	0.3	0.1	0.2	0.1	0.1	0.05

Tabla 2.4: Tareas sintéticas de transferencias dominantes usadas en nuestros benchmarks. Los comandos *HtD*, *K* y *DtH* se definen con una fración de tiempo con respecto a una unidad de tiempo. La unidad de tiempo es 10 ms.

permite especificar tareas cuyo tiempo de transferencia domina sobre su tiempo de computación y viceversa o, incluso, es posible definir tareas con un tiempo de transferencia *HtD* mayor que el tiempo de transferencia *DtH* y a la inversa. Las Tablas 2.3 y 2.4 resumen las diferentes tareas sintéticas diseñadas para validar nuestro modelo. Los valores que aparecen en esta tabla son una fracción de una unidad de tiempo de 10 ms. Por ejemplo, la primera tarea *T0*, toma 1 ms para la etapa *HtD*, 8 ms para la etapa *K* y 1 ms para la etapa *DtH*. La selección de estas tareas pretende reflejar los tiempos de transferencia y computación más habituales en las tareas reales.

Las tareas sintéticas y reales como veremos más adelante se pueden clasificar según la relación entre el tiempo de transferencia y el tiempo de computación. Por lo tanto, tareas donde $t_{HtD} + t_{DtH} > t_k$ son clasificadas como tareas de transferencias dominantes (tareas DT, $T12-23$) y tareas donde $t_{HtD}+t_{DtH} \leq t_k$ son clasificadas como tareas de kernel dominante (tareas DK, $T0-11$). Según esta clasificación, hemos diseñado cinco benchmarks para analizar el modelo que consideran una amplia diversidad de computación y transferencias de datos. La Tabla 2.5 muestra estos benchmarks. Cada benchmark contiene un porcentaje de tareas DK que es incluido en su etiqueta. De esta manera, para el caso de 4 tareas, *BK0* no tiene tareas de kernel dominante (0 %), *BK25* tiene 1 tarea de kernel dominante y 3 tareas de transferencias dominantes (25 %), y así en adelante. Este

Tareas	**4**	**6**	**8**	**16**
BK0	T15, T16, T12, T13	T15, T16, T12, T13, T14, T17	T15, T16, T12, T13 T14, T18, T21, T20	T15: 2, T16: 2, T12: 2, T13: 2, T14: 2, T18: 2, T19: 2, T20: 2
BK25	T0, T12, T15, T14	T0, T1, T16 T12, T13, T14	T0, T1, T2, T15 T16, T12, T13, T14	T0, T1, T2, T7, T15: 2, T16: 2, T12, T13, T14: 2 T13, T18, T21, T20
BK50	T0, T1, T12, T13	T0, T1, T2, T16, T12, T13	T0, T1, T2, T7 T15, T16, T12, T13	T0, T1, T2, T7, T5, T9, T10, T11, T15, T16, T12, T13, T14, T18, T21, T20
BK75	T0, T1, T2, T12	T0, T1, T2, T4, T16, T12	T0, T1, T2: 2, T7 T15, T16, T12	T0: 2, T1: 2, T2: 2, T7: 2 T5, T9: 2, T2, T12, T13: 2, T14
BK100	T0, T1, T2, T3	T0, T1, T3, T2, T5, T6	T0, T1, T2, T7 T5, T9, T10, T11	T0: 2, T1: 2, T2: 2, T7: 2, T5: 2, T9: 2, T10: 2, T11: 2

Tabla 2.5: Benchmarks sintéticos usados. Cada benchmark se define con una etiqueta BKX, donde X es el porcentaje de tareas de kernel dominante en cada benchmark.

sistema de definición de benchmarks puede ser usado con más tareas, por ejemplo con grupos de 6, 8 y 16 tareas, tal y como veremos en capítulos siguientes.

2.5.2. Reales

Las tareas de los benchmarks reales han sido construidos con varios kernels bien conocidos pertenecientes a los SDK de NVIDIA CUDA, OpenCL, AMD OpenCL y Rodinia. La Tabla 2.6 resume los kernels seleccionados junto con su clasificación como kernel dominante o transferencias dominantes. Como se puede observar en la tabla, las tareas DCT, FWT y CONV pueden presentar diferentes comportamientos dependiendo del acelerador utilizado, así como también de los parámetros de entrada y el lenguaje de programación utilizado. Para incrementar la variabilidad de los benchmarks, cada tarea ha sido ejecutada usando varios tamaños de datos de entrada y diferentes parámetros de entrada, que conducen a tiempos de ejecución diferentes. Las Tablas 2.7 y 2.8 muestran, para OpenCL y CUDA respectivamente, los tiempos de ejecución de los comandos de las tareas reales usando diferentes tamaños de datos de entrada.

Las tareas reales han sido combinadas en varios benchmarks de forma similar a la que se utilizó en la Sección 2.5.1 con las tareas sintéticas, pero usando las tareas reales de la Tabla 2.6. Por tanto, en el benchmark *BK0* cada tarea es de transferencias dominantes, el benchmark *BK25* el 25 % de las tareas son de kernel dominante, el benchmark *BK50* el 50 % de las tareas son de kernel dominante y así en adelante.

2.6. Validación del Modelo

Nuestro modelo ha sido evaluado para los entornos de programación CUDA y OpenCL. Para el entorno de programación CUDA hemos utilizado las arquitecturas NVIDIA GPU más modernas hasta el momento, como son Kepler (K20c) y Maxwell (GTX 980), las cuales soportan Hyper-Q. Esta característica hace posible que cada cola de comandos pueda utilizar una cola hardware distinta para lanzar los comandos al acelerador y, por consiguiente, pueda realizar un reordenamiento del lanzamiento de los mismos. Por tanto, para la evaluación de nuestro modelo hemos forzado a que Hyper-Q trabaje con una sola cola hardware, ya que cualquier cambio en el orden de lanzamiento de los comandos no sería modelado y puede introducir un error significativo en nuestro modelo. Este comportamiento se puede conseguir fijando el valor de la variable de entorno $CUDA\_DEVICE\_MAX\_CONNECTIONS = 1$. En cuanto al entorno

Kernel	Descripción	Tipo de Tarea
MM	Matrix Multiplication	DK
BS	Black Scholes	DK
FWT	Fast Walsh Transform	DT/DK
FLW	Floyd Warshall	DK
CONV	Separable Convolution	DK/DT
VA	Vector Addition	DT
TM	Matrix Transposition	DT
DCT	Discrete cosine transform	DT/DK
PAF	Particle Filter	DK
PF	PathFinder	DK

Tabla 2.6: Tareas usadas en los benchmarks reales. Las tareas han sido seleccionadas según su clasificación como de kernel dominante o de transferencias dominantes. Las tareas CONV, DCT y FWT pueden presentar diferente comportamiento según el acelerador y el entorno de programación utilizados. De esta manera, las tareas DCT y FWT pueden ser de transferencias dominantes o de kernel dominante, si son ejecutadas en los aceleradores AMD R9, NVIDIA K20c, o Intel Xeon Phi respectivamente. Similarmente, la tarea CONV puede tener diferente comportamiento dependiendo de sus parámetros de entrada.

Tarea	MM	BS	FWT	FLW	CONV	VA	MT	DCT
OPENCL - AMD R9								
HtD (ms)	0.97-2.57	0.08-1.29	1.29-2.57	0.05-0.07	0.09-0.37	0.65-3.86	2.57-5.15	2.57-5.15
K (ms)	1.80-9.02	2.98-5.57	2.59-5.47	7.77-10.08	1.51-14.58	0.05-0.30	0.29-3.59	0.95-1.89
DtH (ms)	0.14-1.18	0.16-2.17	1.18-2.35	0.09-0.16	0.09-0.37	0.30-1.81	2.36-4.70	2.35-4.71
OPENCL - Intel Xeon Phi								
HtD (ms)	0.36-0.90	0.17-0.63	0.67-1.26	0.03-0.06	0.06-0.17	1.27-7.46	2.58-4.98	1.71-2.25
K (ms)	4.98-5.03	5.25-12.03	4.59-6.39	1.12-9.05	0.56-10.09	0.18-1.18	2.36-1.09	6.97-9.41
DtH (ms)	0.09-0.16	0.33-1.24	0.61-1.21	0.06-0.12	0.17-10.09	0.61-3.68	2.54-4.93	1.67-2.18
OPENCL - NVIDIA K20c								
HtD (ms)	2.51-3.77	0.31-1.25	1.25-5.01	0.01-0.31	0.63-2.53	2.51-12.54	2.60-5.01	2.51-5.01
K (ms)	3.99-7.95	1.25-9.26	1.20-4.94	1.32-9.25	1.47-9.20	0.09-0.44	0.41-2.61	1.55-3.08
DtH (ms)	1.24-2.49	0.62-2.50	1.25-4.98	0.03-0.63	0.62-2.50	1.25-6.19	2.60-4.96	2.48-4.96

Tabla 2.7: Rango de tiempos de ejecución para los comandos *HtD*, *K* y *DtH* pertenecientes a las tareas de los benchmarks reales en OpenCL. El rango de los tiempos de ejecución para cada tarea se obtiene ejecutando la tarea correspondiente con diferentes tamaños de datos de entrada.

Tarea	MM	BS	PF	CONV 1	PAF	VA	TM	FWT	CONV 2
CUDA - NVIDIA K20c									
HtD (ms)	2.53-40.21	0.74-35.97	3.61-3.85	0.33-80.68	0.21-0.18	2.53-241.04	1.27-321.88	2.53-161.06	5.04-10.05
K (ms)	10.60-674.40	8.53-2732.70	7.92-13.56	4.42-139.31	4.13-5.37	0.09-8.53	0.08-24.01	1.30-116.60	0.84-1.67
DtH (ms)	1.26-19.92	0.49-23.99	0.01-0.02	0.33-79.91	0.07-0.08	1.26-119.70	1.27-318.70	1.26-79.86	4.98-9.94
CUDA - NVIDIA GTX 980									
HtD (ms)	1.36-43.48	1.17-19.46	1.94-2.10	0.18-43.47	0.12-0.14	5.41-130.41	2.71-173.86	2.71-86.95	2.72-5.41
K (ms)	2.90-360.95	6.23-2444.32	4.38-9.34	6.52-74.06	2.50-4.13	0.30-7.09	0.21-19.67	2.04-92.52	0.43-0.85
DtH (ms)	0.71-11.15	0.81-13.30	0.01-0.01	0.19-44.51	0.04-0.07	2.79-66.79	2.79-178.07	1.40-44.52	2.79-5.57

Tabla 2.8: Rango de tiempos de ejecución para los comandos *HtD*, *K* y *DtH* pertenecientes a las tareas de los benchmarks reales en CUDA. El rango de los tiempos de ejecución para cada comando se obtienen ejecutando la tarea correspondiente con varios conjuntos de parámetros de entrada. Los kernels *CONV 1* y *CONV 2* corresponden al kernel *CONV* con un número de iteraciones mayor que 1 e igual que 1 respectivamente.

de programación OpenCL, hemos utilizado aceleradores de los fabricantes más relevantes en el mercado, como son AMD R9, NVIDIA K20c e Intel Xeon Phi 5100 series.

Para llevar a cabo la evaluación del modelo, se han utilizado los benchmarks de 4 tareas sintéticas definidas en la Sección 2.5.1 y benchmarks de 4 tareas reales con la misma proporción y características que los benchmarks sintéticos. Para cada experimento se han comprobado todos los posibles órdenes de las tareas para cada benchmark ($4! = 24$ posibles órdenes de tareas). Las Figuras 2.9 y 2.10 muestran la media geométrica del error de predicción obtenido en CUDA, para tareas sintéticas y reales respectivamente. Para entornos de programación CUDA y aceleradores NVIDIA (K20c y GTX980) se han realizado los experimentos para dos tipos de configuraciones: *HyperQ* y *Single Queue*. La configuración de *Single Queue* se obtiene forzando a Hyper-Q para que trabaje con una sola cola hardware. Este comportamiento se puede conseguir utilizando la función de CUDA llamada *cudaStreamSynchronize* para bloquear el hilo CPU hasta que todas las operaciones previamente lanzadas en una determinada CQ se hayan completado, pero esto reduciría la opciones de solapamiento entre diferentes CQs. Una alternativa más apropiada es utilizar eventos para sincronizar comandos o reducir el número colas hardware mediante la variable de entorno $CUDA\_DEVICE\_MAX\_CONNECTIONS = 1$. Ambas figuras, Figuras 2.9 y

2.10, muestran que para la configuración *Single Queue*, nuestro modelo obtiene una media geométrica de error de predicción por debajo del 1.5 %. Sin embargo, cuando se utiliza la configuración *HyperQ* la media geométrica del error de predicción es mayor. Esto es debido a que el acelerador puede cambiar el orden de ejecución de los comandos cuando activamos Hyper-Q con más de una cola hardware (32 colas hardware para este caso). Hyper-Q hace posible que las CQs puedan ejecutarse en un orden diferente al que fueron llamadas, ya que cada CQ emplea una cola hardware distinta para lanzar comandos. Esto puede introducir un error significativo en nuestro modelo, ya que este predice el tiempo total de ejecución de un *TG* dependiendo de un orden específico y, si cambia el orden, empeora la predicción.

Las Figuras 2.11 y 2.12 muestran, para entornos OpenCL, la media geométrica del error de predicción obtenido por benchmark y acelerador, con tareas sintéticas y reales respectivamente. Para entornos de programación OpenCL la característica hardware Hyper-Q no está disponible. Para este tipo de entornos y bencmarks con tareas sintéticas, se puede observar que la media geométrica del error de predicción para todos los benchmarks está por debajo del 1 % para los aceleradores AMD R9 y NVIDIA K20c y 1.12 % para Intel Xeon Phi. Este hecho también se puede observar para el caso de tareas reales representado en la Figura 2.12, donde la media geométrica del error de predicción por benchmark está por debajo del 1 % para los aceleradores AMD R9 y NVIDIA K20c y 1.10 % para Intel Xeon Phi.

2.7. Resumen

En este capítulo hemos presentado un modelo temporal que simula la ejecución de varias tareas independientes en una GPU soportando la concurrencia entre comandos. El modelo permite caracterizar la ejecución de cada uno de los comandos pertenecientes a una tarea, y además simular la interacción de esos comandos con el resto de comandos de otras tareas. Nuestro modelo hace un tratamiento especial en situaciones donde ocurre un solapamiento parcial entre dos transferencias de memoria en direcciones opuestas. El enfoque propuesto en nuestro modelo consigue resultados más precisos en la predicción del tiempo de ejecución de los comandos de transferencias. También hemos diseñado una serie de benchmarks, compuestos tanto por tareas sintéticas como por tareas reales seleccionadas de varios SDKs, que permiten representar cargas de trabajo de distinta naturaleza. Finalmente, hemos validado el modelo usando estos benchmarks con algunas arquitecturas actuales de NVIDIA, como Kepler y Maxwell,

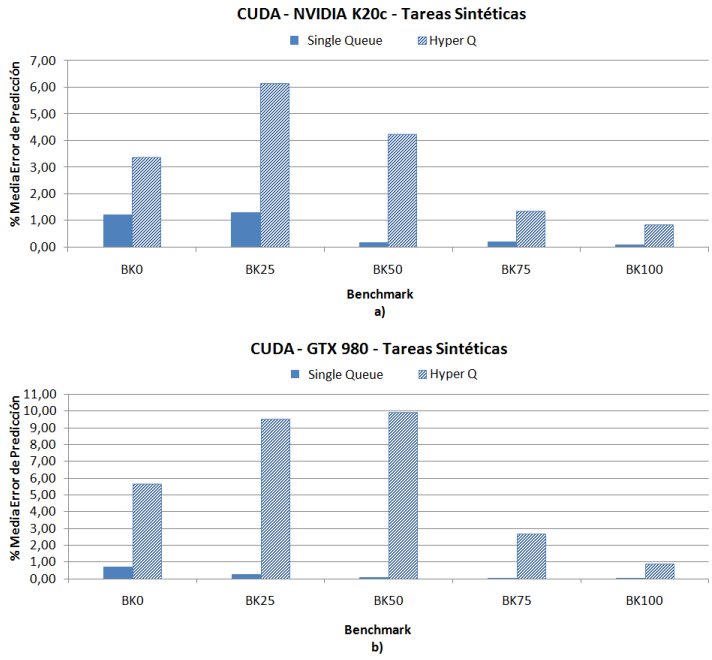


Figura 2.9: Media geométrica del error de predicción para CUDA en los aceleradores NVIDIA K20c (a) y GTX 980 (b), para todas las permutaciones de tareas sintéticas en cada benchmark. Se presentan dos valores: *HyperQ* y *Single Queue*. Hyper-Q es una característica exclusiva de los aceleradores de Nvidia y emplea 32 colas hardware disponibles en el acelerador. La configuración de *Single Queue* solo emplea 1 cola, evita el reordenamiento y deshabilita las transferencias concurrentes en la misma dirección.

de AMD, como GCN, o de Intel, como Xeon Phi, obteniendo un error de simulación inferior al 1.5 %. Las arquitecturas NVIDIA presentan una característica hardware denominada Hyper-Q, la cual puede cambiar el orden de la ejecución de los comandos. Esta característica puede introducir un gran error en la predicción de nuestro modelo, por lo que es necesaria deshabilitarla para mejorar la predicción de este.

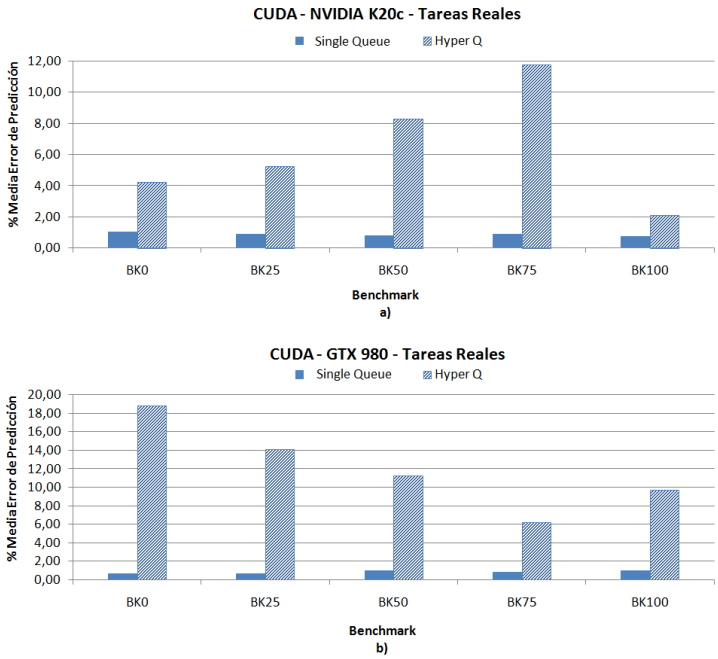


Figura 2.10: Media geométrica del error de predicción para CUDA en los aceleradores NVIDIA K20c (a) y GTX 980 (b), para todas las permutaciones de tareas reales en cada benchmark. Se presentan dos valores: *HyperQ* y *Single Queue*. Hyper-Q es una característica exclusiva de los aceleradores de Nvidia y emplea 32 colas hardware disponibles en el acelerador. La configuración de *Single Queue* solo emplea 1 cola, evita el reordenamiento y deshabilita las transferencias concurrentes en la misma dirección.

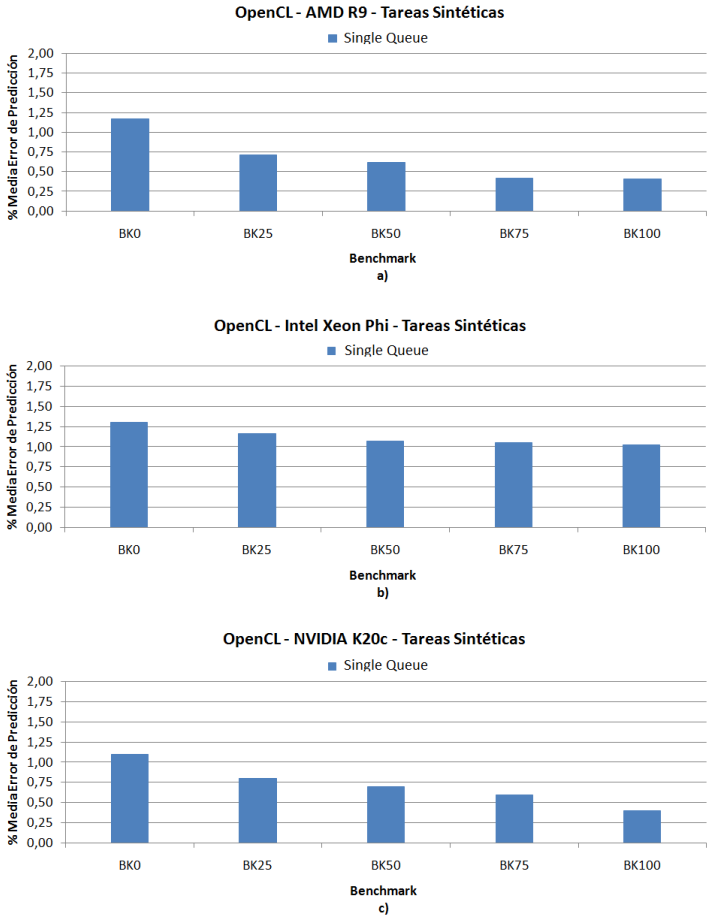


Figura 2.11: Media geométrica del error de predicción para OpenCL en los aceleradores AMD R9 (a), Intel Xeon Phi (b) y NVIDIA K20c (c), para todas las permutaciones de tareas sintéticas en cada benchmark.

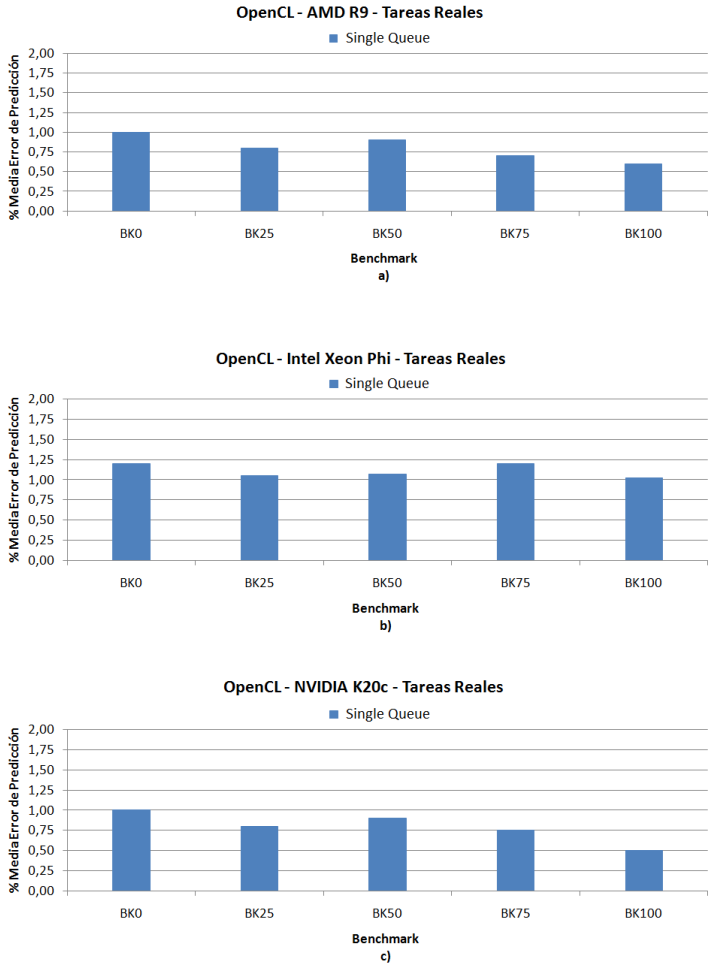


Figura 2.12: Media geométrica del error de predicción para OpenCL en los aceleradores AMD R9 (a), Intel Xeon Phi (b) y NVIDIA K20c (c), para todas las permutaciones de tareas reales en cada benchmark.

3 Planificador Dinámico

En este capítulo se discute como planificar, en tiempo de ejecución, un grupo de tareas con el objetivo de minimizar el tiempo total de ejecución. En capítulos anteriores se ha mostrado que, debido al solapamiento entre comandos pertenecientes a tareas distintas, el orden de ejecución de estos tiene un impacto importante en el tiempo de ejecución de las tareas. Desafortunadamente, el planificador del acelerador no siempre selecciona el mejor orden de ejecución y características hardware como Hyper-Q pueden incluso cambiar el orden inicial de ejecución de los comandos.

En la Sección 3.1 se hace un repaso a algunos de los trabajos más relevantes en el campo de la planificación dinámica de tareas en aceleradores. A continuación, en la Sección 3.2, se presenta un sistema de ejecución de grupos de tareas que permite analizar las tareas que se van a ejecutar en el acelerador y predecir su comportamiento usando el modelo de ejecución que se presentó en la Sección 2.4. Este sistema permite también forzar un orden de ejecución por lo que en la Sección 3.4 se discute una heurística de planificación en tiempo de ejecución capaz de establecer un orden de ejecución de tareas muy cercano al óptimo posible, reduciendo así el tiempo total de ejecución. Por último, el sistema de ejecución y la heurística son validados en la Sección 3.5, tanto para entornos CUDA como OpenCL.

3.1. Estado del arte

Tanto las GPUs como las plataformas MIC se han convertido en unos importantes sistemas de computación de altas prestaciones en diversos campos científi-

cos. Normalmente, este tipo de aplicaciones presentan cargas de trabajo regulares que son bastante apropiadadas para su ejecución en arquitecturas *manycore*. Sin embargo, en problemas donde están presentes cargas de trabajo irregulares se necesitan técnicas de balanceo de carga y planificación dinámica para conseguir un alto rendimiento. Los actuales paradigmas de computación como CUDA y OpenCL no pueden satisfacer aún este tipo de problemas. El balanceo de carga se establece dividiendo el trabajo principal en pequeñas subtareas las cuales son asignadas a las unidades de computación disponibles. La introducción de tareas en GPUs y plataformas MIC es una idea atractiva, debido a la gran cantidad de aplicaciones que presentan este comportamiento, pero la asignación de estas tareas a las unidades de computación requiere de técnicas de planificación dinámica que no son triviales.

La literatura recoge diversos trabajos que se centran en la planificación dinámica de tareas en aceleradores. En [90] estudian diversas técnicas de particionado de los recursos computacionales en GPU para la ejecución concurrente de tareas o *kernels*. Proponen una técnica de particionado orientado a *warps*. Este técnica utiliza un método analítico para calcular el particionado de recursos a través de diferentes *kernels* por medio de ejecuciones previas. El balanceo de carga es un asunto crítico en un sistema. Una buena técnica para realizar un buen balanceo de carga es repartir de forma uniforme las cargas de trabajo entre los procesadores disponibles. Navarro et al. [50] presentan un esquema de planificación dinámica junto con un particionamiento adaptativo, que ajusta el tamaño de datos a computar por cada procesador para prevenir el desbalanceo de carga. En [62] ponen de manifiesto que en sistemas donde CPU y GPU están presentes, la ganancia de rendimiento varía dependiendo de la repartición de recursos seleccionada. En este trabajo los autores afirman que el reparto de recursos computacionales se puede realizar de una forma temporal o espacial, y proponen un esquema dinámico capaz de seleccionar el tipo de reparto por medio de unas medidas de rendimiento de cada *kernel* previamente realizadas. Un reparto espacial de los recursos es también utilizado por Lian et al. [44] para resolver el problema de reparto de recursos entre *kernels* pesados y ligeros en términos de demanda de computación. Cederman et al. [13] investigan sobre GPU la técnica de balanceo de carga denominada robo de tareas o *work stealing*. La idea básica es que si un hilo no tiene tareas para procesar, este puede robar tareas a otros hilos para procesarlas, así como también generar tareas nuevas que son introducidas en un grupo de trabajos propio. Esta técnica también es utilizada en [14], donde introducen un planificador CUDA basado en *work-stealing* para habilitar el paralelismo de tareas y balanceo de carga entre SMs, y en [7] donde la utilizan en escenarios multihilo en CPU. Blumofe et al. [10] estudian los problemas de pla-

nificación de tareas con dependencias, haciendo uso de *work-stealing* y un árbol de dependencias. Este escenario también es investigado en [79] pero haciendo uso de la técnica de hilos persistentes. Los hilos persistentes también son utilizados por Chen et al. [16] para conseguir un balanceo de carga con una granularidad más fina que CUDA. Una técnica similar es la utilizada en [80], donde se utiliza un conjunto de hilos para ejecutar los distintos pasos para ejecutar varios *kernels* concurrentes. En [39] se extiende esta técnica mediante la utilización de eventos, para conseguir el solapamiento entre computación y transferencias de memoria.

Aprovechando la capacidad de algunas nuevas arquitecturas GPU de generar por si solas más trabajo, Wang et al. [85] extienden el actual modelo de ejecución en GPU, para soportar la creación de pequeños bloques de hilos. Este esquema soporta el lanzamiento anidado de bloques de hilos en lugar de *kernels*, para ejecutar de forma paralela distintos elementos de trabajo y conseguir un buen balanceo de carga. Wende et al. [88, 89] utilizan un enfoque de productor-consumidor en escenarios multihilo, para manejar las tareas enviadas a la GPU por distintos hilos en el *host*, presentando también un estudio más amplio en [87] sobre las técnicas de planificación dinámica y balanceo de carga en procesadores masivamente paralelos. Este esquema también ha sido utilizado por Takizawa et al. [76] en entornos OpenCL, donde extienden el mecanismo de eventos OpenCL para la comunicación entre el *host* y el acelerador con el fin de solapar transferencias de datos con computación. En [65], el uso compartido de una o varias GPUs en entornos de virtualización ha sido su estudio principal. En este trabajo los autores extienden el software de virtualización de GPU para su uso compartido. Por otra parte, introducen soluciones al problema de asignar tareas a los distintos procesadores. En particular, utilizan un método de afinidad para seleccionar el procesador más adecuado para la ejecución de una tarea.

Por otra parte, los sistemas heterogéneos donde CPU y GPU están presentes en el mismo chip han surgido en los últimos años como candidatos para mejorar las oportunidades de mejora de rendimiento con el menor uso energético posible [36, 77]. Con el fin de conseguir un buen balanceo entre rendimiento y consumo energético, la planificación de tareas en este tipo de sistemas ha sido el objetivo principal de diversos estudios en los últimos años. Vilches et al. [82] consideran el problema de ejecutar de forma repartida aplicaciones en CPU y GPU en este tipo de sistemas. Los autores proponen una solución que encuentra adaptativamente el mapeo óptimo de computación entre CPU y GPU. Para ello utilizan un modelo analítico que recoge información de rendimiento y consumo de energía en tiempo de ejecución. Luna et al. [27] proponen la utilización de la memoria compartida en sistemas heterogéneos y la utilización del mecanismo *platform atomics* para sincronizar la ejecución entre CPU y GPU. Un mapeo

adaptativo de los recursos de computación denominado Qilin, es también utilizado en [47]. Este esquema se adapta a cambios en el entorno de ejecución, tamaños del problema y configuraciones hardware y software. La búsqueda del mapeo óptimo de computación hacia los distintos procesadores no es una tarea trivial, por lo que existen distintos métodos en la literatura para conducir esta búsqueda. Tanto en [31] como [68], se presenta una heurística para encontrar el mapeo óptimo en sistemas heterogéneos, haciendo uso de un histórico de rendimiento de cada tarea en cada procesador. En [25] se utiliza un planificador dinámico que utiliza características de ejecución de cada tarea, para seleccionar el procesador más adecuado para su ejecución. Por último, en [64] se realiza un estudio más amplio de las diversas técnicas de planificación y entornos de programación disponibles para este tipo de sistemas.

Nuestro principal objetivo es poder planificar eficientemente un conjunto de tareas en un entorno HPC, con aceleradores conectados mediante buses PCIe que son usados concurrentemente por diferentes aplicaciones locales o remotas. De esta manera, buscamos incrementar el uso del acelerador y reducir el tiempo total de ejecución de un conjunto de tareas, explotando las oportunidades de solapamiento entre las mismas. Además, la planificación será llevada a cabo de forma transparente al programador, el cual no tendrá que modificar su código para apreciar la aceleración de sus aplicaciones cuando se ejecutan concurrentemente con otras en el mismo acelerador.

3.2. Planificador Dinámico

Distintos soportes hardware como Hyper-Q (NVIDIA), ACEs (AMD) y los hilos hardware (Intel Xeon Phi) incrementan la capacidad de ejecución de tareas concurrentes en el acelerador, pero dichos soportes no analizan las tareas que están esperando en las colas hardware para mejorar el solapamiento de las tareas concurrentes. El análisis de las tareas residentes en las colas hardware es importante para maximimar el solapamiento entre computación y transferencias de datos. En esta tesis proponemos un mecanismo en tiempo de ejecución que lleva a cabo este análisis. Este emplea un hilo proxy CPU que se encarga de lanzar varias tareas independientes pertenecientes a diferentes *workers* (hilos productores CPU). Todos los *workers* envían la información de las tareas, según las funciones de CUDA u OpenCL, a un buffer que es constantemente consultado por el hilo proxy. Esta información contiene datos como:

- Identificador del *worker*.

- Identificador de la tarea.
- Lista de Comandos. Una lista con las llamadas a la funciones de transferencia de memoria y de lanzamiento de los comandos *kernels*.
- Lista de Número de Parámetros. Una lista con el número de parámetros que necesita cada comando de la tarea.
- Lista de Parámetros. Una lista con los parámetros que necesita cada comando de la tarea.
- Lista de Cantidad de Datos. Una lista con las cantidades de datos a transmitir por la tarea, tanto desde CPU a GPU como viceversa.

Esto se representa en la Figura 3.1, donde se muestran tres *workers*, Wj ($j = 0, 1, 2$), enviando 4 tareas $T_{i.j}$ ($i = 0, 1, 2, 3$). Por ejemplo, el *worker* $W0$ envía las tareas $T_{0,0}$, $T_{1,0}$, $T_{2,0}$ y $T_{3,0}$, las cuales son agrupadas en un buffer compartido indicado como $APIs_{0,0}$ en la figura.

Por otra parte, tal y como se muestra en la Figura 3.2, una vez que el hilo proxy detecta alguna tarea en el buffer compartido, construye un grupo de tareas, (TG), para ser lanzadas en el acelerador. El hilo proxy puede aplicar alguna heurística de reordenamiento al TG que permita maximizar la concurrencia entre tareas, para luego enviar los comandos *HtD*, *K* y *DtH* de las tareas a sus correspondientes colas de comandos. Hemos asumido que las tareas enviadas por un mismo *worker* mantienen, de forma natural, una dependencia entre ellas ya que probablemente representarán las distintas etapas en las que se ha descompuesto una aplicación. De esta manera, la tarea $T_{1,0}$ enviada por el *worker* $W0$, no puede ser ejecutada antes de que termine la tarea $T_{0,0}$ del mismo *worker*. Para mantener esta dependencia, el hilo proxy crea una lista de eventos de las últimos comandos de cada tarea (normalmente comandos *DtH*) ejecutados. Esta lista es compartida entre el hilo proxy y un hilo de sincronización encargado de activar las tareas cuyas dependencias han sido resueltas. Los eventos de esta lista son insertados junto con los comandos *DtH* para indicar cuando estos han finalizado. De esta manera, en la Figura 3.2 el evento *Evt* $DTH_{0,0}$- *W0* hace referencia al evento insertado junto con el comando $DtH_{0,0}$ perteneciente al *worker* 0. La lista de eventos de los últimos comandos *DtH* es constantemente consultada por el hilo de sincronización, para detectar si algún evento ha cambiado su estado. La Figura 3.3 muestra la lista de eventos en el instante de tiempo t donde ningún evento ha sido alcanzado (eventos con recuadro en color rojo). En el estado $t + 1$ el comando $DtH_{0,1}$ ha sido completado y por lo tanto el evento *Evt* $DtH_{0,1}$- *W1* cambia su estado a alcanzado (recuadro verde en la lista de eventos). Este cambio

de estado en el evento de la lista es detectado por el hilo de sincronización, que activa la siguiente tarea del *worker* $W1$ en el buffer compartido.

El esquema propuesto también es apropiado para mejorar el mecanismo de planificación de tareas en sistemas distribuidos donde los procesos remotos pueden compartir un acelerador [20] [37] [41]. De forma similar, el hilo proxy también se puede usar en modelos de programación para arquitecturas heterogéneas donde el *host* y el acelerador residen en el mismo chip (StarPU [8] XKaapi [24] y OmpSs [21]). Por tanto, este escenario, aparte de su aplicabilidad en otros sistemas, nos permite centrarnos en el objetivo principal de este trabajo, que es el análisis del impacto del reordenamiento de tareas en la ejecución concurrente de las mismas. En las arquitecturas NVIDIA como Kepler y sus sucesoras, está presente la característica denominada *paralelismo dinámico* [59]. Esta característica hace posible que un propio *kernel* o tarea genere a su vez *subkernels* o subtareas. Esta característica no ha sido estudiada en esta tesis y por tanto es una limitación presente del modelo propuesto.

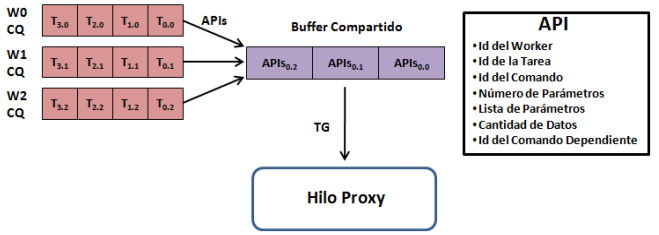


Figura 3.1: Los hilos productores (*workers*) insertan la información de las tareas en un buffer compartido junto con el hilo proxy. Este buffer es consultado por el hilo proxy para recoger las tareas a ejecutar y realizar su planificación.

3.3. Definición de los Comandos

El planificador dinámico explicado en la sección 3.2 tendrá que encargarse también de realizar las distintas llamadas a CUDA u OpenCL, pertenecientes a los comandos de las distintas tareas. Por tanto, tiene que ser capaz de lanzar las operaciones de los comandos asociados a la tarea específica a ejecutar en cada momento, pero sin modificar el código específico de cada tarea. Para ello, se ha hecho uso de tres propiedades de los lenguajes orientados a objetos (OO): herencia, polimorfismo y funciones virtuales [73].

En primer lugar, cada tarea se define mediante una clase. Por un lado, los

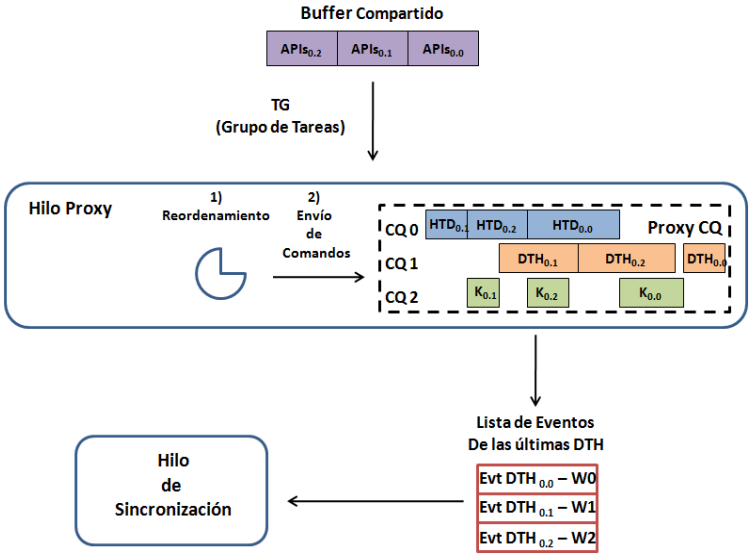


Figura 3.2: El hilo proxy detecta las tareas listas en el buffer compartido y construye un *TG* de tareas a ejecutar. Las tareas del *TG* son reordenadas y se lanzan sus respectivos comandos a las CQs correspondientes. Los comandos son enviados al acelerador por el hilo proxy empleando tres CQs (un acelerador con dos motores DMA). Los eventos de los últimos comandos *DtH* son recogidos en una lista compartida entre el hilo proxy y un hilo de sincronización.

atributos de la clase serán las variables, tanto en el *host* como en el acelerador, necesarias para la ejecución de la tarea. Por otro lado, los métodos de esta clase recogerán las distintas llamadas a CUDA u OpenCL necesarias para los comandos de la tarea. La tabla 3.1 muestra los principales métodos que debe de tener la clase de una tarea y su definición. Teniendo en cuenta la definición de los métodos de la clase que representará a una tarea, el código CUDA u OpenCL de la misma debe organizarse para ser encapsulado en cada uno de los métodos recogidos en la tabla 3.1, sin ser modificado. Una vez definida la clase, y organizado y encapsulado el código de los comandos en los métodos de la tabla 3.1, el planificador solo tiene que realizar llamadas a los métodos correspondientes según las operaciones que quiera realizar. Por ejemplo, si quiere lanzar un comando *HtD* realizará una llamada al método *memHtDAsync*, si quiere lanzar un comando *K* realizará una llamada al método *launchKernelAsync*, y así con todas las operaciones y comandos.

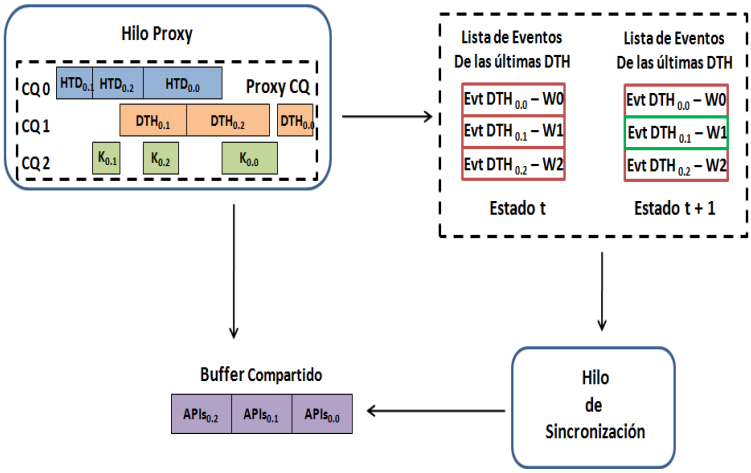


Figura 3.3: El hilo proxy crea una lista de eventos insertados junto con los comandos *DtH*. Esta lista es compartida con un hilo de sincronización. Los eventos de esta lista indican cuando los comandos *DtH* han finalizado. Cuando el hilo de sincronización detecta un cambio de estado en algún evento de esta lista, este activa para su ejecución en el buffer compartido, la siguiente tarea del *worker* correspondiente.

Método	Definición
allocHostMemory	Reserva memoria en el host
freeHostMemory	Libera memoria en el host
initData	Genera o inicializa los datos de la tarea en memoria del host
allocDeviceMemory	Reserva memoria en el acelerador
freeDeviceMemory	Libera memoria en el acelerador
memHtDAync	Realiza las llamadas CUDA u OpenCL para lanzar transferencias HtD asíncronas
memDtHAsync	Realiza las llamadas CUDA u OpenCL para lanzar transferencias DtH asíncronas
launchKernelAsync	Realiza las llamadas CUDA u OpenCL para lanzar comandos kernel asíncronos

Tabla 3.1: Definición de los métodos de la clase que representa a una tarea.

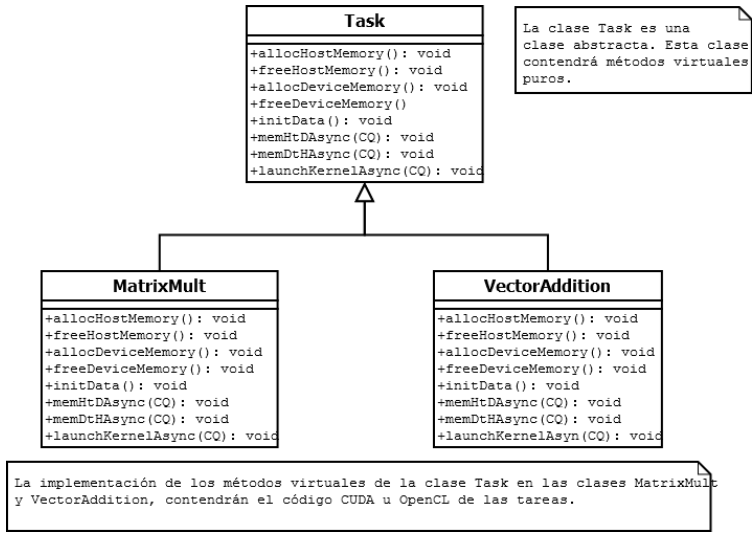


Figura 3.4: Ejemplo de diagrama de clases para la definición de las tareas MatrixMult y VectorAddition. La clases de las tareas *MatrixMult* y *VectorAddition* son hijas de una clase abstracta llamada *Task*. La clase *Task* contiene métodos virtuales puros que definen la interfaz que tienen que implementar sus clases hijas.

El planificador, en un momento determinado, tendrá varios objetos de clases de tareas distintas y, por tanto, debe ser capaz de diferenciar entre los métodos de un objeto y de otro. Para ello, aplicamos la propiedad de herencia de los lenguajes OO y realizamos una generalización de las clases de las tareas a una clase denominada *Task*. La figura 3.4 muestra un ejemplo de diagrama de clases, para las tareas *MatrixMult* y *VectorAddition*. La clases de las tareas *MatrixMult* y *VectorAddition* son hijas de una clase abstracta llamada *Task*. La clase *Task* contiene métodos virtuales puros. Los métodos virtuales puros de una clase abstracta no contienen una implementación, porque su implementación se encuentra en las clases que heredan de la clase abstracta. Es decir, estos métodos solo proporcionan una interfaz que deben cumplir las clases hijas. Por tanto, las clases *MatrixMult* y *VectorAddition* contienen los mismos métodos que la clase padre abstracta *Task*, pero con la implementación específica para cada clase de tarea. En el Código 3.1 se muestra un extracto del código C++/CUDA del cuerpo de los métodos de la clase que representa a la tarea *MatrixMult*. De esta manera, se proporciona al planificador un conjunto común de métodos para lanzar los comandos. El planificador solo realizará llamadas a los métodos de la clase *Task*. Las llamadas a los métodos virtuales de la clase *Task* se traducirán, por medio de la propiedad de

polimorfismo, en las correspondientes llamadas a los métodos de las clases hijas. En el Código 3.2 se puede ver un extracto del código C++/CUDA encargado de llamar a los distintos métodos de las tareas mostrados en la Tabla 3.1. En primer lugar se crea un vector llamado *tasks_v* que contendrá todas las tareas a planificar en un determinado momento (línea 3, Código 3.2). Este vector se crea utilizando la librería STL de C++ y contendrá punteros a clases abstractas *Task* para hacer uso de la característica de polimorfismo comentada anteriormente. A continuación, creamos las tareas con sus datos de inicialización y las introducimos en el vector *tasks_v* (líneas 6-12, Código 3.2). Una vez introducidas las tareas en el vector de tareas, recorreremos el vector *tasks_v* llamando a los métodos de cada una de sus tareas, para reservar memoria en el *host* y en la GPU y también inicializar los datos (líneas 17-22, Código 3.2). A continuación, recorreremos de nuevo el vector para llamar a los métodos de las tareas que se encargan de ejecutar sus comandos y lanzarlos por el stream CUDA correspondiente (líneas 26-31, Código 3.2). Una vez lanzados los comandos de las tareas por su stream CUDA correspondiente, debemos sincronizar el host con la ejecución de la GPU (línea 34, Código 3.2). Por último, recorremos de nuevo el vector *tasks_v* para llamar a los métodos de liberación de memoria de cada una de las tareas (líneas 37-41, Código 3.2).

```
1  void MatrixMult::allocHostMemory(void)
2  {
3    cudaMallocHost((void **)&h_A_MM, mem_size_A_MM * sizeof(float));
4    cudaMallocHost((void **)&h_B_MM, mem_size_B_MM * sizeof(float));
5    cudaMallocHost((void **)&h_C_MM, mem_size_C_MM * sizeof(float));
6  }
7  void MatrixMult::freeHostMemory(void)
8  {
9    if(h_A_MM!=NULL)          cudaFreeHost(h_A_MM);
10   if(h_B_MM!=NULL)          cudaFreeHost(h_B_MM);
11   if(h_C_MM!=NULL)          cudaFreeHost(h_C_MM);
12 }
13
14 void MatrixMult::allocDeviceMemory(void)
15 {
16   cudaMalloc((void**) &d_A_MM, mem_size_A_MM * sizeof(float));
17   cudaMalloc((void**) &d_B_MM, mem_size_B_MM * sizeof(float));
18   cudaMalloc((void**) &d_C_MM, mem_size_C_MM * sizeof(float));
19 }
20 void MatrixMult::freeDeviceMemory(void)
21 {
22   if(d_A_MM!=NULL) cudaFree(d_A_MM);
23   if(d_B_MM!=NULL) cudaFree(d_B_MM);
24   if(d_C_MM!=NULL) cudaFree(d_C_MM);
25 }
26 void MatrixMult::initData(void)
```

```
{
  for (int i = 0; i < mem_size_A_MM; ++i)
        h_A_MM[i] = 1.0f;
    for (int i = 0; i < mem_size_B_MM; ++i)
        h_B_MM[i] = 0.01f;

  cudaMemset(d_C_MM, 0, mem_size_C_MM * sizeof(float));

}
void MatrixMult::memHtDAsync(cudaStream_t stream)
{
  cudaMemcpyAsync(d_A_MM, h_A_MM, mem_size_A_MM*sizeof(float),
    cudaMemcpyHostToDevice, stream);
  cudaMemcpyAsync(d_B_MM, h_B_MM, mem_size_B_MM*sizeof(float),
    cudaMemcpyHostToDevice, stream);

}
void MatrixMult::memDtHAsync(cudaStream_t stream)
{
  cudaMemcpyAsync(h_C_MM, d_C_MM, mem_size_C_MM*sizeof(float),
    cudaMemcpyDeviceToHost, stream);
}
void MatrixMult::launchKernelAsync(cudaStream_t stream)
{
  dim3 threadsS(blockSize, blockSize);
  dim3 blocksS(uiWC_MM/ threadsS.x, uiHC_MM / threadsS.y);

  matrixMul<<< blocksS, threadsS, 0, stream >>>(d_C_MM,
                          d_A_MM,
                          d_B_MM, uiWA_MM, uiWB_MM);
}
```

Código 3.1: Código C++/CUDA para la implementación de los métodos de la tarea MatrixMult.

```

//Vector de tareas a ejecutar
vector <Task *> tasks_v;

//Creamos las tareas
int sizes[4];
sizes[0] = 1024; sizes[1] = 1024;
sizes[2] = 1024; sizes[3] = 1024;
tasks_v.push_back(new MatrixMult(sizes));

int vector_size = 4194304;
tasks_v.push_back(new VectorAddition(vector_size));

//Numero de tareas insertadas
int n_tasks = 2;

```

```
17 for(int i = 0; i < n_tasks; i++)
18 {
19     tasks_v.at(i)->allocHostMemory();
20     tasks_v.at(i)->initData();
21     tasks_v.at(i)->allocDeviceMemory();
22 }
23
24
25 //Lanzamos las tareas por los streams
26 for(int i = 0; i < n_tasks; i++)
27 {
28     tasks_v.at(i)->memHtDAsync(streams[i]);
29     tasks_v.at(i)->launchKernelAsync(streams[i]);
30     tasks_v.at(i)->memDtHAsync(streams[i]);
31 }
32
33 //Sincronizamose el host con las tareas lanzadas en la GPU.
34 cudaDeviceSynchronize();
35
36 //Liberamos memoria de las tareas
37 for(int i = 0; i < n_tasks; i++)
38 {
39     tasks_v.at(i)->freeHostMemory();
40     tasks_v.at(i)->freeDeviceMemory();
41 }
```

Código 3.2: Código C++/CUDA para las llamadas a los métodos de las clases *MatrixMult* y *VectorAddition* empleando clases abstractas y polimorfismo.

3.4. Heurística de Planificación

La búsqueda de un orden óptimo de ejecución cuando un *TG* es enviado al acelerador debe ser realizada en tiempo de ejecución. Consecuentemente, los métodos basados en enfoques de fuerza bruta no son aplicables, ya que el número de combinaciones posibles para un *TG* de N tareas es $N!$. En esta sección proponemos una heurística que intenta minimizar la duración de los tiempos de inactividad en el acelerador cuando se ejecuta un *TG*. Estos tiempos de inactividad se pueden reducir cuando se encuentra un orden de ejecución óptimo, de forma que la mayoría de comandos de transferencias sean ejecutados concurrentemente con otros comandos *kernel*. Para conseguir este propósito se propone el Algoritmo 1.

La heurística empieza con la selección de la primera tarea (Algoritmo 1 línea: 2). Esta tarea se selecciona del conjunto de tareas pendientes por ordenar (*Remaining Tasks*, *RT*). Esta tarea será aquella que tenga un tiempo de transferencia

HtD pequeño y un tiempo de *kernel* K suficientemente grande comparado con las tareas restantes. De esta manera se reducen al comienzo los períodos de inactividad en el acelerador y, al mismo tiempo, aparecen más opciones de solapamiento para las tareas posteriores. Si existen varias tareas que cumplan estos requisitos, entonces la tarea con mayor tiempo de transferencia *DtH* es seleccionada para mejorar la concurrencia entre esa transferencia y los comandos *kernel* de las tareas posteriores. La primera tarea seleccionada, T_{ini}, se añade al conjunto de tareas ordenadas (*Ordered Tasks*, OT) y borrada del conjunto RT (líneas: 3-4). A continuación, se actualizan las variables t_HTD, t_K y t_DTH por medio de la simulación del conjunto OT (línea: 5).

Algoritmo 1 Algoritmo de Reordenamiento de un TG.

RT = $\{T_1, ..., T_N\}$: Conjunto de Tareas Pendientes por Ordenar.
OT = $\{\phi\}$: Conjunto de Tareas Ordenadas.
t_HTD= 0: Tiempo de Finalización del último comando HTD.
t_K= 0: Tiempo de Finalización del último comando K.
t_DTH= 0: Tiempo de Finalización del último comando DTH.

1: **Procedimiento** Reordenamiento de Tareas
2: $T_{ini} = seleccion\_primera\_tarea(RT)$
3: $OT = OT \cup \{T_{ini}\}$
4: $RT = RT \setminus \{T_{ini}\}$
5: $[t\_HTD, t\_K, t\_DTH] = actualizar(OT)$
6: **mientras** $|RT| > 2$ **hacer**
7: $T_{sig} = seleccion\_siguiente\_tarea(RT, t\_HTD, t\_K, t\_DTH)$
8: $OT = OT \cup \{T_{sig}\}$
9: $RT = RT \setminus \{T_{sig}\}$
10: $[t\_HTD, t\_K, t\_DTH] = actualizar(OT)$
11: **fin mientras**
12: $[T_{penultima}, T_{ultima}] = seleccion\_ultimas\_tareas(RT, OT)$
13: $OT = OT \cup \{T_{penultima}, T_{ultima}\}$
14: **fin**

Mientras el número de tareas en el conjunto RT sea mayor que 2 se seleccionan las siguientes tareas (líneas: 6-11). Esta selección es llevada a cabo mediante una función, *seleccion_siguiente_tarea* (línea: 7), que se encarga de buscar el mejor ajuste entre el comando K de la tarea anteriormente seleccionada y el comando *HtD* de la nueva tarea, y entre los comandos *DtH* de las tareas previamente seleccionadas y el comando K de la nueva tarea. Más concretamente, nuestro modelo de ejecución se usa para predecir el tiempo de computación de los actuales

comandos en el conjunto OT, compararlos con el tiempo de ejecución de las nuevas tareas, y maximimizar el grado de solapamiento entre los comandos.

Cuando el número de tareas en el conjunto RT es igual a 2, se lleva a cabo la selección de la última tarea por medio de la función *seleccion_ultimas_tareas* (línea: 12). Esta funciona como la función *seleccion_siguiente_tarea* pero añadiendo un nuevo criterio basado en la duración del comando DtH, para evitar un largo período de inactividad durante la ejecución del comando DtH de la última tarea.

Durante la planificación de la ejecución concurrente de un TG hay que tener en cuenta las restricciones sobre la cantidad total de memoria global disponible en el acelerador. La ejecución concurrente de un TG puede que necesite reservar más cantidad de memoria global en el acelerador de la que sería necesaria durante la ejecución secuencial de las tareas del mismo. Esto es debido a que todas las tareas necesitan, inicialmente, reservar memoria para sus datos de entrada y de salida porque no puede liberarse memoria durante la ejecución del TG, ya que la liberación de la memoria requiere que todos los comandos precedentes terminen de ejecutarse y, por tanto, ningún comando posterior del TG podría ejecutarse concurrentemente.

Consecuentemente, el máximo de memoria global disponible puede ser un requisito adicional para seleccionar las tareas que forman parte de un TG. Una solución eficiente y flexible para este problema necesitará del desarrollo de una política específica de reserva de memoria que sustituya al *runtime* de la GPU. Este aspecto no se estudia en el ámbito de esta tesis sino que se deja como una línea futura de investigación. Por lo tanto, se asume que hay suficiente memoria global disponible para reservar los datos de entrada y salida de todas las tareas del TG.

3.5. Validación

En esta sección se evalúa la fiabilidad y utilidad de nuestro modelo de ejecución de grupos de tareas y de la heurística de planificación en escenarios multihilo. En estos escenarios varios hilos (*workers*) ejecutan aplicaciones que envían una o varias tareas al acelerador. Todos los *workers* envían la información de la tarea a realizar mediante las funciones pertenecientes a la interfaz de programación utilizada. En esta tesis hemos utilizado dos de las interfaces de programación más habituales, como son CUDA y OpenCL. Por lo tanto, la validación de nuestros modelos se llevará a cabo tanto para CUDA como para OpenCL.

3.5.1. Resultados para Entornos NVIDIA-CUDA

Para la evaluación de nuestro modelo en entornos NVIDIA-CUDA, hemos utilizado aceleradores de algunas de las arquitecturas más avanzadas hasta el momento, como son Kepler (K20c) y Maxwell (GTX980), las cuales soportan la característica denominada Hyper-Q. Esta característica aumenta el número de colas hardware disponibles (hasta 32) para lanzar comandos al acelerador. De esta forma, Hyper-Q asocia cada CQ a una cola hardware distinta, y cada cola hardware es planificada de forma independiente.

Todas las pruebas consideran diferentes benchmarks compuestos por varios *kernels* que se han seleccionado para representar cargas de trabajo lo más realistas posibles, siguiendo las directrices presentadas en la Sección 2.5. Más concretamente, estos *kernels* han sido obtenidos a partir de las implementaciones proporcionadas por NVIDIA-CUDA SDK y Rodinia [15] tal y como se especifica en la Tabla 3.2, donde se indican la procedencia del código y su clasificación como kernel dominante (DK) o de transferencias dominantes (DT). Cada tarea ha sido ejecutada utilizando varios conjuntos de parámetros de entrada para incrementar la variabilidad de los benchmarks. La Tabla 2.8 en la Sección 2.5.2 muestra el rango de los tiempos de ejecución de los comandos de las tareas usando conjuntos diferentes de parámetros de entrada.

Kernel	Fuente	Descripción	Tipo de Tarea
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 3.2: Tareas usadas en los benchmarks reales para entornos CUDA. Estas tareas han sido seleccionadas según su clasificación de kernel dominante (DK) o de transferencias dominantes (DT). La tarea *CONV* se puede clasificar como de kernel dominante o transferencias dominantes dependiendo de sus parámetros de entrada.

Para comprobar la fiabilidad y utilidad de nuestro modelo y de la heurística de planificación hemos llevado a cabo varios experimentos. El primer experimento consiste en la ejecución de cuatro tareas independientes usando para ello cuatro CQs (una tarea por cada CQ). La Tabla 3.3 muestra las tareas empleadas en

# Tareas	Benchmark	Descripción
4	BK0	VA: 1, TM: 1, FWT: 2
	BK25	MM: 1, VA: 1, TM: 1, FWT: 1
	BK50	MM: 1, BS: 1, VA: 1, TM: 1
	BK75	MM: 1, BS: 1, PF: 1, VA: 1
	BK100	MM: 2, BS: 1, CONV1: 1

Tabla 3.3: Composición de tareas para cada benchmark en los experimentos de cuatro tareas, en entornos CUDA.

cada benchmark. Cuando en un benchmark aparece más de una vez la misma tarea, se debe a que esta ha sido utilizada con distintos conjuntos de datos de entrada. Todas las permutaciones posibles para cada conjunto de tareas se han ejecutado 15 veces, y se ha calculado el tiempo de ejecución medio para cada permutación. El objetivo de este experimento es, por un lado, confirmar la fiabilidad de nuestra heurística de planificación y, por otro lado, compararla con los resultados obtenidos cuando Hyper-Q se encarga de repartir las tareas entre las distintas colas hardware. Por tanto, todos las pruebas se han ejecutado usando las dos configuraciones ya introducidas, *Single Queue* e *Hyper-Q.*

En la configuración de ejecución *Hyper-Q*, varias colas hardware están disponibles mediante la inicialización de la variable de entorno *CUDA_DEVICE_MAX_CONNECTIONS* a 32. Esto hace que, cuando varias CQs envían tareas al acelerador, el orden exacto de ejecución depende del orden de lanzamiento, el número de motores DMA y la característica hardware Hyper-Q. Esta característica permite a la unidad GMU manejar múltiples colas hardware para reducir o eliminar falsas dependencias, pero también conlleva una alteración en el orden de lanzamiento de las tareas. En cambio, en la configuración de ejecución *Single Queue*, una sola cola hardware está disponible porque la variable de entorno *CUDA_DEVICE_MAX_CONNECTIONS* se inicializa a 1 y, por tanto, el acelerador respeta el orden de lanzamiento de las tareas.

Para cada una de las configuraciones se han ejecutado cada benchmark con todas las posibles permutaciones y, a continuación, se ha anotado el mejor y peor tiempo de ejecución, y también se ha calculado el tiempo medio de ejecución. Además, el orden de tareas predicho por nuestra heurística también ha sido ejecutado usando la configuración *Single Queue* para recoger su tiempo de ejecución y así poder compararla con los tiempos obtenidos usando cualquiera de las dos configuraciones. Todos estos tiempos de ejecución se muestran en la Tabla 3.4. Los datos obtenidos demuestran la fiabilidad de nuestra heurística, ya que se puede comprobar que se consigue, de forma consistente, tiempos cercanos al mínimo de cualquiera de las dos configuraciones de ejecución, y siempre por debajo de

GPU	K20c					GTX 980				
Benchmark	BK0	BK25	BK50	BK75	BK100	BK0	BK25	BK50	BK75	BK100
Tiempo Max. Hyper-Q (ms)	34.15	39.47	41.38	42.54	71.41	36.41	35.42	31.95	31.05	32.68
Tiempo Min. Hyper-Q (ms)	31.24	32.17	28.20	29.57	69.91	34.44	32.91	26.10	22.33	31.64
Tiempo Medio Hyper-Q (ms)	32.74	35.95	33.06	34.96	70.76	35.58	34.19	28.72	25.70	32.14
Tiempo Max. Single Queue (ms)	32,62	39,01	40,94	45,08	75,90	31,48	31,72	30,63	28,18	29,66
Tiempo Min. Single Queue (ms)	30,78	31,74	29,18	30,63	68,64	27,22	26,72	23,01	20,65	26,89
Tiempo Medio Single Queue (ms)	31.71	34.89	34.95	38.22	71.42	29.03	29.27	25.53	24.02	28.08
Tiempo Heur. (ms)	31,49	32,13	29,52	30,63	69,95	27,29	27,47	23,15	21,86	26,89

Tabla 3.4: Tiempos de ejecución de los benchmarks compuestos por cuatro tareas reales, para las configuraciones *Single Queue* e *Hyper-Q*. Todas las permutaciones posibles en el orden de las tareas han sido ejecutadas para recoger el tiempo máximo, mínimo y medio de todas ellas. También se ha recogido el tiempo de ejecución para el orden de tareas predicho por nuestra heurística.

los tiempos medios para cualquier benchmark y configuración.

Para evaluar la utilidad de nuestro modelo y de la heurística, se muestran en la Figura 3.5 los speedup de todos los tiempos de ejecución obtenidos previamente con respecto al peor tiempo de ejecución para la configuración *Single Queue*. Además, se han incluido resultados para benchmarks con tareas sintéticas como los presentados en la Sección 2.5.1. Se puede ver que, para todos los benchmarks, nuestra heurística obtiene un speedup muy cercano al máximo. Además, el speedup de la heurística es siempre mejor que el speedup medio. Es importante observar que, incluso en escenarios donde hay poco margen de mejora gracias a la concurrencia porque hay pocas transferencias (por ejemplo, en *BK100*), aún así se consigue algo de aceleración seleccionando como primera tarea aquella con menor tiempo de *HtD* y como última tarea la que tenga menor tiempo de *DtH*. En aquellos benchmarks en los que hay una mezcla más equilibrada de tareas DK y DT el margen de mejora es más elevado y, por tanto, resulta más útil usar una heurística que permita obtener un orden de planificación cercano al óptimo.

De forma similar, la Figura 3.6 muestra el speedup de los tiempos de ejecución con respecto al peor tiempo de ejecución alcanzado con la configuración *Hyper-Q*. Además, se ha incluido el speedup conseguido con nuestra heurística usando

la configuración *Single Queue*, con respecto al peor tiempo de ejecución con la configuración *Hyper-Q*. Después de analizar los resultados, podemos señalar que, usando la configuración *Hyper-Q*, hay variaciones de hasta un 45 % entre diferentes órdenes para el mismo benchmark (variaciones entre el speedup máximo y mínimo), mientras que nuestra heurística siempre obtiene un speedup mejor que el speedup medio y a la vez muy cercano al valor óptimo. Además, en algunos benchmarks usando la GPU GTX980, nuestra heurística algunas veces consigue un speedup mayor que el conseguido por la mejor permutación usando la configuración *Hyper-Q*. Esto es debido a que a veces dos transferencias en el mismo sentido se pueden llegar a ejecutar de forma concurrente alternando bloques de comunicación. En el caso de producirse este hecho, se retrasan los comandos de *kernel* asociados a esas tareas impidiendo que se solapen con otros comandos de transferencia. En cambio, la configuración *Single Queue* no permite que dos transferencias en el mismo sentido puedan ejecutarse concurrentemente por lo que este problema no aparece.

Aunque hay casos donde el mejor orden en la configuración *Hyper-Q* obtiene un tiempo de ejecución menor que el orden seleccionado por nuestra heurística para la configuración *Single Queue*, es importante señalar que estos órdenes donde se obtiene mayor rendimiento no pueden ser predichos de antemano. En resumen, nuestra heurística consigue una mejora consistente para cada benchmark y configuración, mientras que los resultados usando la configuración *Hyper-Q* pueden variar ampliamente, produciendo tiempos de ejecución peores en muchas ejecuciones, por lo que nuestra heurística resulta útil para asegurar un rendimiento cercano al óptimo en todas las circunstancias.

El siguiente experimento se ha llevado a cabo para comprobar la escalabilidad de nuestra heurística con respecto al tiempo de ejecución de las tareas. En este caso, se han utilizado conjuntos de datos de mayor tamaño y, consecuentemente, se han incrementado los tiempos de ejecución de las tareas reales. En la Tabla 3.5 se muestran los tiempos de ejecución máximo, mínimo y medio para las configuraciones *Hyper-Q* y *Single Queue* en una GPU K20c, así como también el tiempo de ejecución conseguido con nuestra heurística cuando se utiliza la configuración *Single Queue*. La Figura 3.7 muestra el speedup de estos tiempos de ejecución con respecto al peor tiempo de ejecución conseguido con la configuración *Hyper-Q* (izquierda) y con respecto al peor tiempo conseguido por la configuración *Single Queue* (derecha). De esta manera, para grandes conjuntos de datos, se observa que nuestra heurística también obtiene para cada benchmark, speedups muy cercanos al máximo y mejor que el speedup medio. Es importante señalar que los mejores speedups se consiguen cuando hay varias tareas de kernel dominante y de transferencias dominantes (por ejemplo: $BK25$, $BK50$ y $BK75$), ya que hay

más oportunidades de solapamiento.

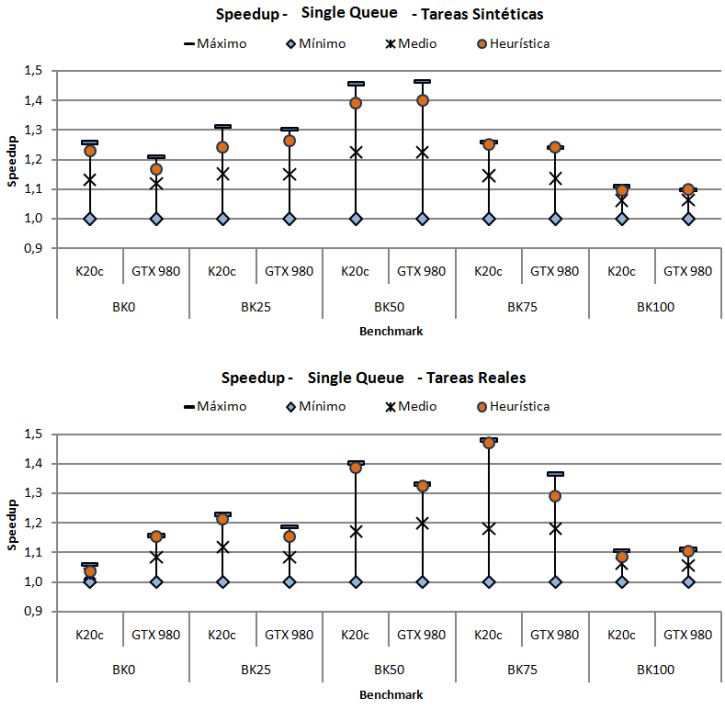


Figura 3.5: Speedup conseguido con tareas sintéticas (arriba) y tareas reales (abajo), para cada benchmark y acelerador, con respecto a la peor permutación. Todos los experimentos han sido desarrollados usando la configuración de ejecución *Single Queue*. El speedup máximo es conseguido por la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio y el speedup de la heurística se calcula usando el orden obtenido por nuestra heurística.

El siguiente experimento intenta reproducir un escenario real en el que uno o más procesos (o hilos) ejecutan parte de su computación en la CPU (o *host*) y otra parte en el acelerador. Además, cada uno puede ejecutar muchas tareas diferentes en el acelerador. De esta manera, en cualquier instante de tiempo, puede haber una cantidad variable de tareas esperando a ser ejecutadas por el acelerador. Para reproducir esta situación, se ha estudiado un escenario donde los benchmarks anteriores son ejecutados mediante T hilos que aleatoriamente ejecutan tareas de un benchmark específico. En la Tabla 3.6 se muestra la composición de tareas reales de cada benchmark como una función del número de

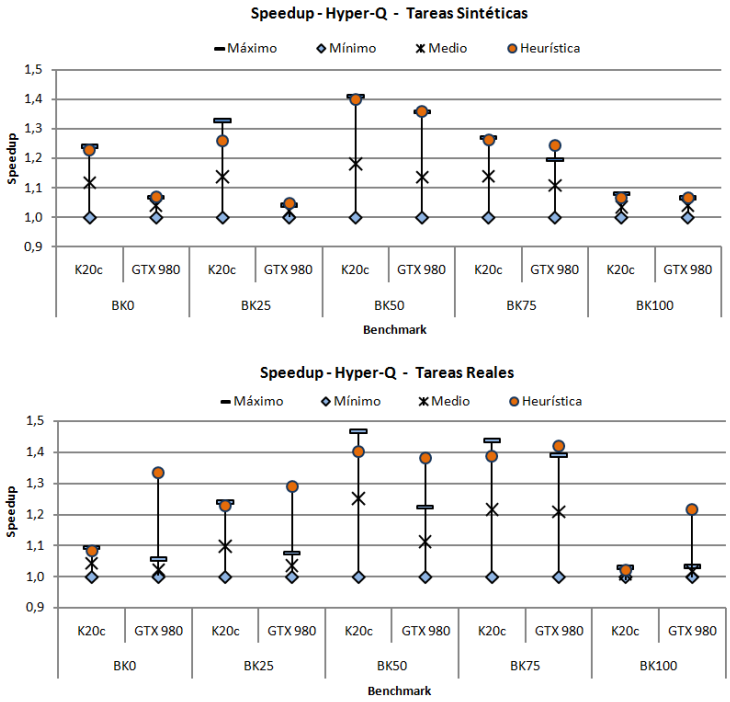


Figura 3.6: Speedup conseguido con tareas sintéticas (arriba) y tareas reales (abajo), para cada benchmark y acelerador, con respecto a la peor permutación. Todos los experimentos han sido desarrollados usando la configuración de ejecución *Hyper-Q*. El speedup máximo es conseguido por la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio y el speedup de la heurística se calcula usando el orden obtenido por nuestra heurística, cuando se ejecuta con la configuración *Single Queue*.

GPU	K20c				
Benchmark	BK0	BK25	BK50	BK75	BK100
Tiempo Max. Hyper-Q (ms)	1412.15	1326.61	1465.36	1718.23	1606.69
Tiempo Min. Hyper-Q (ms)	1276.16	1144.00	986.84	1431.18	1550.28
Tiempo Medio Hyper-Q (ms)	1348.55	1220.95	1095.08	1509.84	1563.67
Tiempo Max. Single Queue (ms)	1345.69	1341.34	1415.05	1685.23	1653.59
Tiempo Min. Single Queue (ms)	1215.07	1073.46	1006.89	1431.55	1537.12
Tiempo Medio. Single Queue (ms)	1290.23	1193.30	1129.92	1538.55	1584.22
Tiempo Heur. (ms)	1267.17	1075.87	1048.10	1440.58	1551.89

Tabla 3.5: Tiempos de ejecución máximo, mínimo y medio usando conjuntos de datos mayores, para los benchmarks de tareas reales en la GPU K20c, para las configuraciones *Hyper-Q*, *Single Queue* y nuestra heurística.

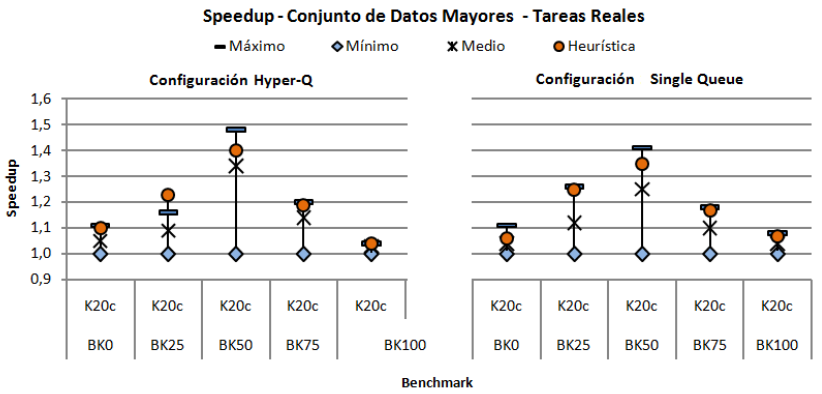


Figura 3.7: Speedups conseguidos usando los tiempos de la Tabla 3.5, con respecto a la peor permutación usando la configuración *Hyper-Q* (izquierda) y la peor permutación usando la configuración *Single Queue* (derecha).

# Tareas	4	8	16
Benchmark		NVIDIA K20c	
BK0	VA: 1, TM: 1, FWT: 2	VA: 2, TM: 2, FWT: 2, CONV2: 2	FWT: 1, TM: 9, VA: 5, CONV2: 1
BK25	MM: 1, VA: 1, TM: 1, FWT:1	MM: 2, BS: 1, VA: 2, TM: 2, FWT: 1	MM: 2, BS: 1, CONV1: 1, TM: 6, VA: 5
BK50	MM: 1, BS: 1 VA: 1, TM: 1	MM: 1, BS: 1, PF: 1, CONV1: 1, TM: 2, VA: 1, FWT: 1	MM: 6, BS: 1, CONV1: 1, FWT: 1, TM: 5, VA: 2
BK75	MM: 1, BS: 1 PF: 1, VA: 1	MM: 1, BS: 1, PF: 1, PAF: 1, CONV1: 1, VA: 1, TM: 1, FWT: 1	MM: 10, BS: 1, CONV1: 1, TM: 2, FWT: 1, VA: 1
BK100	MM: 2, BS: 1, CONV1: 1	MM: 2, BS: 3, CONV1: 3	MM: 14, BS: 1, CONV1: 1
		GTX 980	
BK0	VA: 1, TM: 1, FWT: 2	CONV2: 4, TM: 2, VA: 2	CONV2: 2, TM: 9, VA: 5
BK25	MM: 1, VA: 1, TM: 2	MM: 1, CONV1: 2, VA: 1 TM: 2, CONV2: 2	MM: 2, BS: 1, CONV1: 1, TM: 6, VA: 6
BK50	MM: 1, BS: 1, TM: 1, VA: 1	MM: 1, BS: 1, CONV1: 2, TM: 2, VA: 1, CONV2: 1	MM: 6, BS: 1, CONV1: 1, CONV2: 1, TM: 5, VA: 2
BK75	MM: 1, BS: 1, PF: 1, VA: 1	MM: 1, BS: 1, PF: 1, CONV1: 2, VA: 1, TM: 2	MM: 10, CONV1: 1, BS: 1, CONV2: 1, TM: 2, VA: 1
BK100	MM: 2, BS: 1, CONV1: 1	MM: 2, BS: 3, CONV1: 3	MM: 4, BS: 6, CONV1: 6

Tabla 3.6: Composición de tareas para cada benchmark empleando, 4, 8 y 16 tareas.

GPU	# CQs	Benchmarks Sintéticos					Benchmarks Reales				
		BK0	BK25	BK50	BK75	BK100	BK0	BK25	BK50	BK75	BK100
K20c	4	1.08	1.09	1.13	1.10	1.05	1.06	1.14	1.18	1.11	1.08
	6	1.06	1.12	1.16	1.12	1.04	1.07	1.09	1.13	1.10	1.01
	8	1.06	1.12	1.14	1.11	1.03	1.06	1.16	1.11	1.15	1.00
	16	1.08	1.10	1.10	1.02	1.01	1.02	1.09	1.13	1.04	1.01
GTX 980	4	1.09	1.11	1.15	1.11	1.05	1.09	1.09	1.14	1.19	1.13
	6	1.07	1.14	1.15	1.12	1.04	1.05	1.07	1.14	1.19	1.03
	8	1.07	1.13	1.15	1.13	1.03	1.04	1.15	1.16	1.17	1.00
	16	1.07	1.09	1.11	1.03	1.01	1.02	1.05	1.08	1.12	1.03

Tabla 3.7: Resultados para los benchmarks con tareas sintéticas y reales, usando 4, 6, 8 y 16 CQs, en las GPUs K20c y GTX980. Los valores mostrados son la media geométrica de las aceleraciones de nuestra heurística ejecutada con la configuración *Single Queue*, con respecto a la ejecución con la configuración *Hyper-Q*.

# Tareas	4	6	8	16
Tiempo Medio de Planificación CPU (ms)	0.07	0.13	0.22	1.04
Tiempo Medio de Ejecución GPU (ms)	25.10	33.54	45.54	90.95

Tabla 3.8: Tiempo medio de planificación empleado por el hilo proxy ejecutándose en una CPU Intel Core(TM)2 Quad, para benchmarks sintéticos usando 4, 6, 8 y 16 tareas. Además se muestra el tiempo de ejecución medio en una GPU K20c de un grupo de 4, 6, 8 y 16 tareas.

tareas. Cada hilo se asocia a una CQ diferente y lanza un total de $N = 40$ tareas, con un retraso aleatorio entre lanzamientos consecutivos. Este retraso puede ser 0 (no existe retraso) para simular una carga computacional fuerte, o seleccionado aleatoriamente de una distribución uniforme con valores entre 0 y D, $\mathcal{U}(0, D)$, para simular una carga computacional variable. D ha sido seleccionado como la mitad de los tiempos de ejecución medios de los *kernels*, para asegurarnos de que hay alguna tarea para ejecutar la mayoría del tiempo. De esta forma, cada experimento ha sido llevado a cabo usando $T = 4$, 6, 8 y 16 hilos, sin retraso y con un retraso $\mathcal{U}(0, D)$. Cada uno de los experimentos ha sido ejecutado 15 veces y se ha calculado la mediana de los tiempos para eliminar resultados espurios. Para permitir una comparación justa entre nuestra heurística y la configuración *Hyper-Q*, hemos empleado las mismas tareas y valores de retraso aleatorio para

ambas configuraciones. La Tabla 3.7 muestra la media geométrica[1] de los speedups obtenidos para cada benchmark de tareas sintéticas y reales en las GPUs K20c y GTX980.

Se puede ver que nuestra heurística obtiene siempre una mejora de rendimiento con respecto a la configuración *Hyper-Q*. Esta mejora en el rendimiento depende del número de CQs y de la composición de los benchmarks, es decir, la proporción entre tareas de kernel dominante y de transferencias dominantes. Los mejores resultados se obtienen cuando hay una mezcla heterogénea de estas tareas, ya que en estos casos el orden puede ser importante para solapar transferencias de unas tareas con los *kernels* de otras. De esta manera, en los benchmarks $BK0$ y $BK100$ la mayoría del tiempo de ejecución es consumido por comandos de transferencias y comandos de *kernels*, respectivamente, y por tanto, el orden de lanzamiento no tiene un impacto importante en el tiempo total de ejecución y la aceleración obtenida es baja. Por otra parte, a medida que el número de CQs se incrementa y, consecuentemente, el número de colas hardware disponibles, Hyper-Q dispone de más comandos de tareas independientes. El incremento de comandos puede permitir a Hyper-Q llevar a cabo mejores decisiones de planificación y reducir la desventaja con respecto a nuestra heurística.

Hemos medido también el tiempo de ejecución empleado por el hilo proxy durante la aplicación de la heurística. En la Tabla 3.8 se muestra este tiempo junto con la media del tiempo de ejecución de un grupo de tareas concurrentes. Se puede observar que el tiempo de ejecución empleado en aplicar la heurística se incrementa a medida que aumenta el número de tareas, ya que el espacio de búsqueda para la heurística se hace mayor. Aún así, el tiempo de ejecución empleado en la aplicación de la heurística se mantiene por debajo del 1.2 %, con respecto al tiempo de ejecución de un grupo de tareas en el peor de los casos (16 tareas).

Debido a los retrasos aleatorios, el tiempo total de ejecución depende del instante de tiempo en que las últimas tareas son lanzadas, lo que podría incluso ocasionar la aparición de períodos de inactividad en la GPU. Para obtener una comprensión más profunda del rendimiento de nuestro modelo, hemos anotado cuantos comandos concurrentes (*HtD*, *K*, *DtH*) se han ejecutado. La Figura 3.8 muestra el porcentaje de tiempo donde hay uno, dos y tres comandos concurrentes durante la ejecución del experimento. Nuestra heurística alcanza un alto porcentaje de tiempo donde tres comandos concurrentes se están ejecutando al mismo tiempo, es decir, la GPU es completamente utilizada durante mayor tiempo. Este hecho, se aprecia mejor con tareas reales, donde la configuración *Hyper-Q*

[1] La media geométrica es normalmente usada para comparar speedups.

casi nunca es capaz de ejecutar tres comandos concurrentemente, mientras que nuestro método consigue este objetivo durante más del 10 % del tiempo.

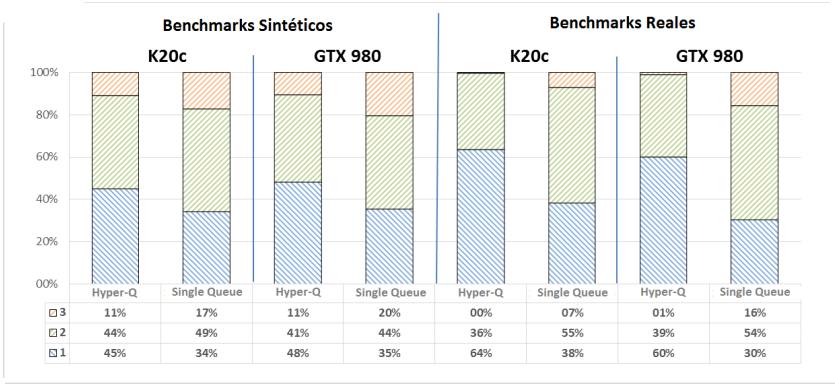


Figura 3.8: Resultados de concurrencia para benchmarks sintéticos (izquierda) y benchmarks reales (derecha). Cada columna representa el porcentaje de tiempo en que se están ejecutando concurrentemente uno, dos y tres comandos, para las GPUs K20c y GTX 980, usando las configuraciones de *Hyper-Q* y *Single Queue*.

3.5.2. Resultados para Entornos OpenCL

Para la evaluación de nuestro modelo en entornos OpenCL, hemos utilizado aceleradores de diferentes tipos como son NVIDIA K20c, AMD R9 e Intel Xeon Phi 5100 series. De esta manera, podemos comprobar la validez de nuestros modelos en diferentes arquitecturas. Las pruebas se han realizado sobre una implementación del sistema proxy descrito en la Sección 3.2, y se han usado varios benchmarks compuestos de tareas reales, junto con benchmarks de tareas sintéticas como los presentados en la Sección 2.5. Más concretamente, los *kernels* de las tareas reales han sido seleccionados de las implementaciones en OpenCL proporcionadas por NVIDIA-CUDA SDK, AMD SDK e Intel SDK. La Tabla 3.9 lista las tareas seleccionadas junto con su clasificación como de kernel dominante (DK) o de transferencias dominantes (DT). Cada tarea ha sido ejecutada utilizando varios conjuntos de parámetros de entrada para incrementar la variabilidad de los benchmarks. La Tabla 2.7 en la Sección 2.5.2 muestra el rango de los tiempos de ejecución de los comandos de las tareas, usando conjuntos diferentes de parámetros de entrada.

Al igual que en la sección anterior, tanto nuestro modelo de ejecución de ta-

Kernel	Descripción	Tipo de Tarea
MM	Matrix Multiplication	DK
BS	Black Scholes	DK
FWT	Fast Walsh Transform	DT/DK
FLW	Floyd Warshall	DK
CONV	Separable Convolution	DK
VA	Vector Addition	DT
TM	Matrix Transposition	DT
DCT	Discrete cosine transform	DT/DK

Tabla 3.9: Tareas usadas en los benchmarks reales para entornos OpenCL. Las tareas han sido seleccionadas según su clasificación de kernel dominante o de transferencias dominantes. Las tareas DCT y FWT pueden tener diferente comportamiento según el acelerador utilizado. De esta manera, ambas tareas pueden ser de transferencias dominantes o de kernel dominante, si son ejecutados en el acelerador AMD R9 y NIVIDA K20c, o Intel Xeon Phi respectivamente.

reas como nuestra heurística han sido evaluados en un escenario multihilo. En los experimentos realizados hemos considerado un conjunto de T tareas independientes, donde T toma valores de 4, 6 y 8. En cada conjunto, un lote de N tareas dependientes estará disponible, donde N puede tomar valores de 1, 2 o 4. Las $T \cdot N$ tareas se seleccionan aleatoriamente de los correspondientes benchmarks sintéticos o reales. Para establecer la bondad del orden determinado por la heurística comparado con todos los posibles órdenes, hemos definido 2 configuraciones de ejecución. Estas configuraciones se denominan configuración *Sin Reordenamiento* y configuración *Heurística*.

Configuración *Sin Reordenamiento*. Un hilo envía asíncronamente los comandos pertenecientes a un conjunto de $T \cdot N$ tareas, teniendo en cuenta las dependencias impuestas por tareas pertenecientes al mismo lote. A diferencia de [40], donde solamente se emplea una CQ para enviar todos los comandos de computación, en nuestro enfoque hacemos posible CKE lanzando cada comando de computación por una CQ diferente. En cada experimento se seleccionan aleatoriamente $T \cdot N$ tareas y se llevan a cabo 15 ejecuciones de todas las posibles permutaciones de tareas ($(T!)^N$). Por último, se mide el tiempo de ejecución medio para cada permutación. Hay que indicar que no se aplica ningún reordenamiento a las tareas que forman parte de una permutación dada. En el caso de que en esta configuración se quiera habilitar CKE, se utilizarán los esquemas de lanzamiento mostrados en las Figuras 2.2(b) y 2.3(b). De lo contrario, se utilizarán los esquemas de lanzamiento definidos en las Figuras 2.2(a) y 2.3(a).

Configuración *Heurística*. Esta configuración considera T hilos lanzando N tareas consecutivas por hilo. Para cada experimento se seleccionan las mismas tareas que para la configuración *Sin Reordenamiento*. Los hilos escriben las correspondientes llamadas a las funciones OpenCL para lanzar las tareas en un buffer compartido. Las dependencias entre las tareas lanzadas por un mismo hilo son forzadas, imponiendo que una nueva tarea no puede ser escrita en el buffer hasta que la tarea anterior no haya terminado su ejecución. De esta manera, el número máximo de tareas concurrentes en un TG es T. El hilo proxy lee el buffer compartido y aplica la heurística para calcular el mejor orden de planificación para ese TG. A continuación, se envían al acelerador los comandos de las tareas reordenadas. Por último, una vez que el hilo proxy envía el comando *HtD* de la última tarea pertenciente al actual TG, consulta de nuevo el buffer compartido y repite el ciclo. Las tareas reordenadas son ejecutadas 15 veces para obtener su tiempo de ejecución medio y evitar tiempos espurios. Esta configuración siempre será utilizada sin habilitar CKE, por tanto se utilizarán los esquemas de lanzamiento definidos en las Figuras 2.2(a) y 2.3(a).

Por un lado las Figuras 3.9, 3.10 y 3.11 muestran los resultados obtenidos para los benchmarks sintéticos en los acelerador AMD R9, NVIDIA K20c y Intel Xeon Phi respectivamente. Por otro lado, las Figuras 3.12, 3.13 y 3.14 representan los resultados obtenidos por los benchmarks reales en los aceleradores AMD R9, NVIDIA K20c e Intel Xeon Phi respectivamente. En este experimento se habilita CKE para la configuración *Sin Reordenamiento*, mientras que para nuestra heurística no. Los resultados muestran el speedup conseguido por la media geométrica (símbolo de cruz) y el mínimo (rectángulo azul) de los tiempos de ejecución de la configuración *Sin Reordenamiento*, con respecto al tiempo de ejecución máximo (rombo azul con speedup igual a 1) de la misma configuración. De esta forma podemos visualizar el rango de valores de speedup para todos los posibles órdenes de tareas en la configuración *Sin Reordenamiento* (segmento vertical con un rectángulo y rombo azul en sus extremos). Se han evaluado todas las posibles permutaciones para la configuración *Sin Reordenamiento* usando 4 hilos ($T = 4$) y $N = 1$, 2 y 4. En el caso de $T = 6$, se han ejecutado todas las permutaciones para $N = 1$, sin embargo, para $N = 2$ solamente se ha ejecutado un subconjunto elegido aleatoriamente, del 5 % de todas las permutaciones. Para $T = 8$, N solamente toma el valor de 1 debido a la gran cantidad de permutaciones disponibles para $N > 1$. Para el caso del acelerador Xeon Phi, los experimentos han sido solamente ejecutados para $N = 1$ debido a que tiene solo un motor DMA ($N = 2$ y $N = 4$ producen los mismos valores de speedup). Además, se indica el speedup conseguido por el orden calculado por nuestra heurística (círculo naranja) con respecto al tiempo de ejecución máximo para la configuración *Sin*

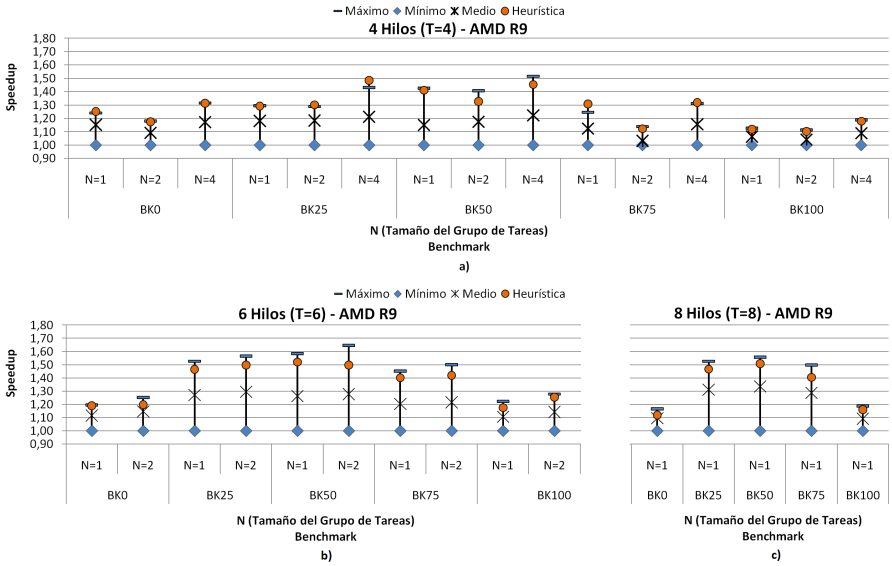


Figura 3.9: Speedups con CKE conseguidos en el acelerador AMD R9, con tareas sintéticas y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística.

Reordenamiento.

Para los benchmarks sintéticos, las Figuras 3.9, 3.10 y 3.11 muestran, como era de esperar, que el impacto del reordenamiento de tareas es mayor para los benchmarks $BK25$, $BK50$ y $BK75$ ya que se consiguen mayores speedups. La razón de este comportamiento es que estos benchmarks contienen tareas de diferente tipo (de kernel y transferencias dominantes) por lo que se pueden encontrar mejores oportunidades de solapamiento de comandos. Además, se puede observar que nuestra heurística predice la mayoría de las veces órdenes muy cercanos a la mejor permutación para cualquier benchmark y valores de T y N, y siempre mejor que el tiempo medio de ejecución conseguido con la configuración *Sin Reordenamiento*. Hay casos en AMD R9 (por ejemplo: $BK25$, $T = 4$, $N = 4$) y Xeon Phi (por ejemplo: $BK50$, $T = 6$, $N = 1$) donde nuestra heurística obtiene mejor rendimiento que la mejor permutación de la configuración *Sin Reordenamiento*. Después de un estudio de estos resultados específicos hemos llegado a la

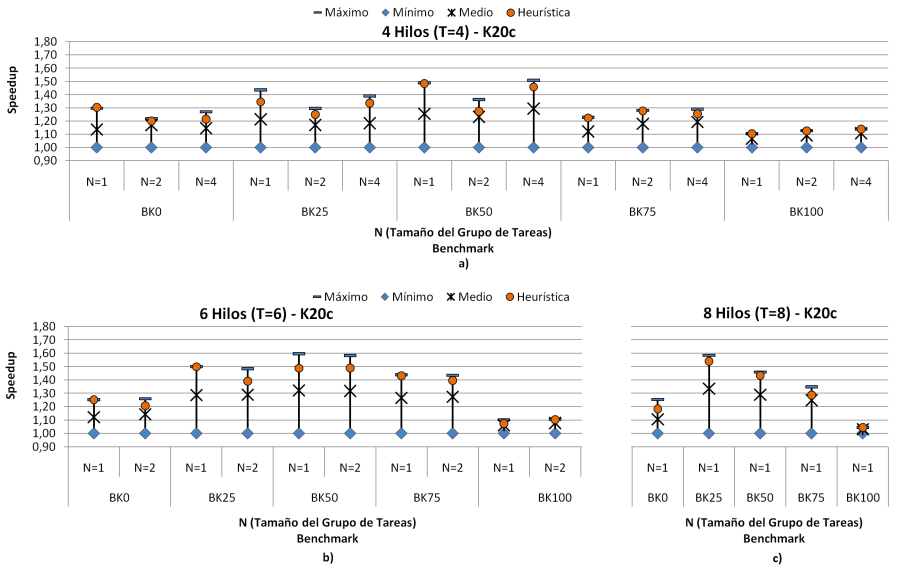


Figura 3.10: Speedups con CKE conseguidos en el acelerador NVIDIA K20c, con tareas sintéticas y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística.

conclusión de que estos aceleradores, AMD y Xeon Phi, son capaces de distribuir sus recursos hardware entre dos kernels que por separado agotarían alguno de esos recursos, y por tanto el uso de CKE incrementa su tiempo de ejecución con respecto a una configuración donde no se habilita CKE. De forma adicional, hemos evaluado también el tiempo de ejecución empleado al aplicar la heurística cuando se emplean diferentes valores de T (número de tareas concurrentes). La Tabla 3.10 muestra el tiempo empleado por la heurística (tiempo medio de planificación en CPU) y el tiempo medio de ejecución de las tareas concurrentes después de aplicar la heurística (tiempo medio de ejecución en GPU). Se puede ver que la penalización provocada por el tiempo de planificación está siempre por debajo del 0.4 %.

Conclusiones similares se pueden extraer de los resultados de los benchmarks reales mostrados en las Figura 3.12, 3.13 y 3.14. Nuestra heurística es capaz siempre de mejorar el valor medio obtenido por la configuración *Sin Reordenamiento*

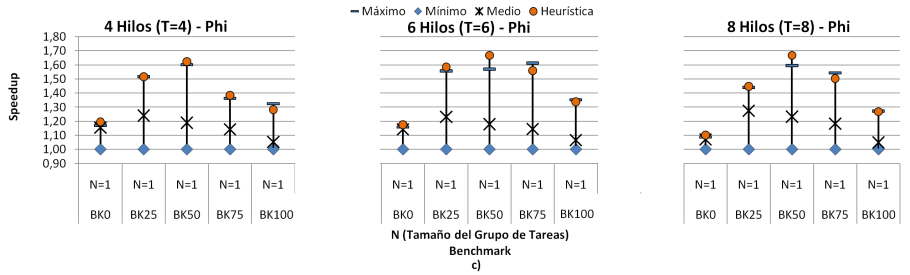


Figura 3.11: Speedups con CKE conseguidos en el acelerador Intel Xeon Phi, con tareas sintéticas y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística.

y muchas veces es capaz de alcanzar un speedup muy cercano al mejor valor de esta configuración. Hay algunos casos donde, aunque el speedup conseguido por la heurística es mejor que el valor medio, se encuentra muy alejado del mejor valor posible (por ejemplo: $BK50$ con $T = 4$ y $N = 4$ para la GPU NVIDIA K20c). A pesar de eso, el speedup medio conseguido por la heurística para todos los casos es todavía muy alto como se muestra en la Figura 3.18. De esta manera, para el acelerador AMD R9 nuestra heurística obtiene un speedup medio de 1.23 que es el 96 % de la mejora obtenida por el mejor orden de la configuración *Sin Reordenamiento* (1.24). Finalmente, los valores obtenidos para Xeon Phi 1.16 (86 %), y K20c, 1.27 (87 %) indican que nuestra heurística es capaz de alcanzar un orden muy cercano al óptimo.

Un último experimento ha sido llevado a cabo para un escenario donde no se habilita CKE para la configuración *Sin Reordenamiento*. Las Figuras 3.15, 3.16 y 3.17 muestran los resultados obtenidos con benchmarks reales en los aceleradores AMD R9, NVIDIA K20c e Intel Xeon Phi respectivamente. Los resultados muestran el speedup conseguido por la media geométrica (símbolo de cruz) y el mínimo (rectángulo azul) de los tiempos de ejecución de la configuración *Sin Reordenamiento* con respecto al tiempo de ejecución máximo (rombo azul). En este experimento la configuración *Sin Reordenamiento* ha sido utilizada con el tipo de lanzamiento de tareas mostrado en la Figuras 2.2(a) y 2.3(a). En estos esquemas de lanzamiento no se habilita CKE ya que todos los comandos de computación son lanzados por la misma CQ. A partir de estos resultados, se pue-

***T* (Número Concurrente de Tareas)**	4	6	8
Tiempo Medio de Planificación en CPU (ms)	0.06	0.10	0.22
Tiempo Medio de Ejecución en GPU (ms)	28.04	37.82	49.78

Tabla 3.10: Tiempo medio de planificación en CPU empleado por el hilo proxy ejecutándose en una CPU Intel Core 2 Quad para benchmarks con 4, 6 y 8 tareas sintéticas. El tiempo medio de ejecución en GPU también se muestra para un grupo de tareas de 4, 6 y 8 tareas sintéticas en la GPU NVIDIA K20c.

den extraer conclusiones similares a las ya obtenidas, donde nuestra heurística predice la mayoría de las veces órdenes muy cercanos a la mejor permutación y siempre mejor que el tiempo de ejecución medio.

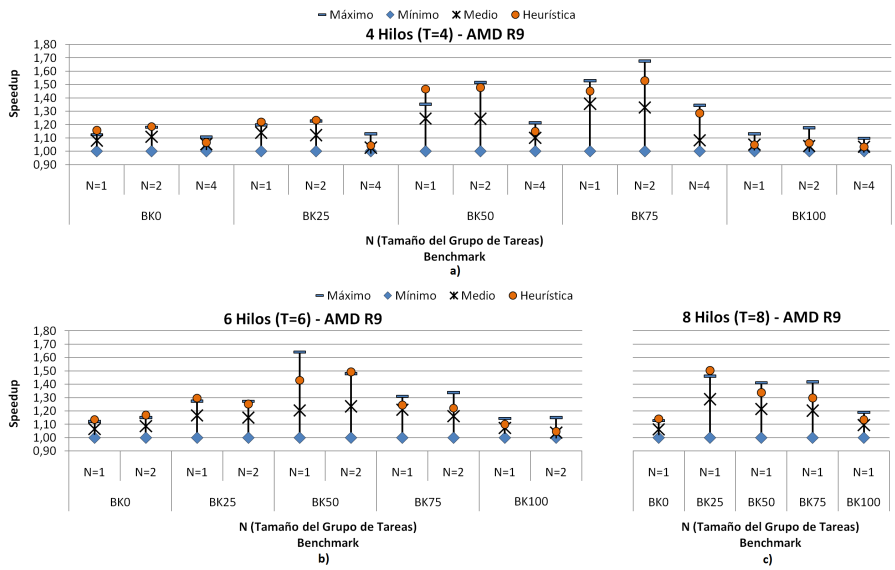


Figura 3.12: Speedups con CKE conseguidos en el acelerador AMD R9, con tareas reales y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística.

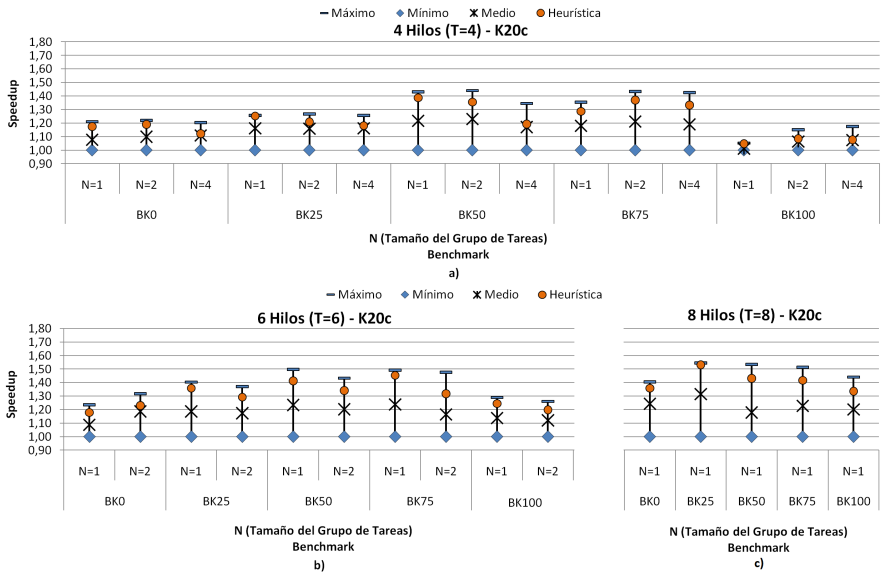


Figura 3.13: Speedups con CKE conseguidos en el acelerador NVIDIA K20c, con tareas reales y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística.

3.6. Resumen

En este capítulo hemos presentado un sistema de ejecución de grupos de tareas, que permite analizar las tareas que se van a ejecutar en el acelerador y predecir su comportamiento usando el modelo de ejecución que se presentó en el capítulo anterior. Este sistema permite también forzar un orden de ejecución por lo que hemos diseñado una heurística de planificación, capaz de funcionar en tiempo de ejecución, que puede establecer un orden de ejecución de tareas muy cercano al óptimo posible, reduciendo así el tiempo total de ejecución. Además, este sistema es capaz de lanzar las distintas operaciones de los comandos asociados a una tarea específica, sin modificar el código de la tarea en si. Por último, hemos evaluado el sistema de ejecución y la heurística para determinar su fiabilidad, aplicabilidad y escalabilidad, comparando los valores obtenidos por nuestro sistema con la aplicación de una búsqueda de fuerza bruta del orden óptimo de

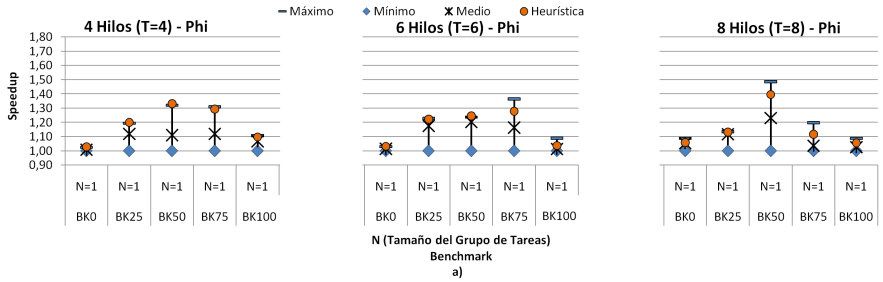


Figura 3.14: Speedups con CKE conseguidos en el acelerador Intel Xeon Phi, con tareas reales y para cada benchmark. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística.

planificación. Esta evaluación se ha llevado a cabo tanto para entornos NVIDIA-CUDA como para entornos OpenCL. En entornos NVIDIA-CUDA se ha forzado que nuestro sistema trabaje con Hyper-Q desactivado (empleando una sola cola hardware) y sin ejecución concurrente de *kernels*, CKE. Aunque nuestro sistema deshabilita ambas características, hemos demostrado que realiza una planificación más eficiente que cuando Hyper-Q y CKE están activadas. En entornos OpenCL la característica hardware Hyper-Q no está disponible pero sí CKE. Nuestro sistema ha sido evaluado frente a situaciones donde tanto CKE está habilitada como en las que no, demostrando en ambas que la planificación realizada consigue mejores resultados de ejecución.

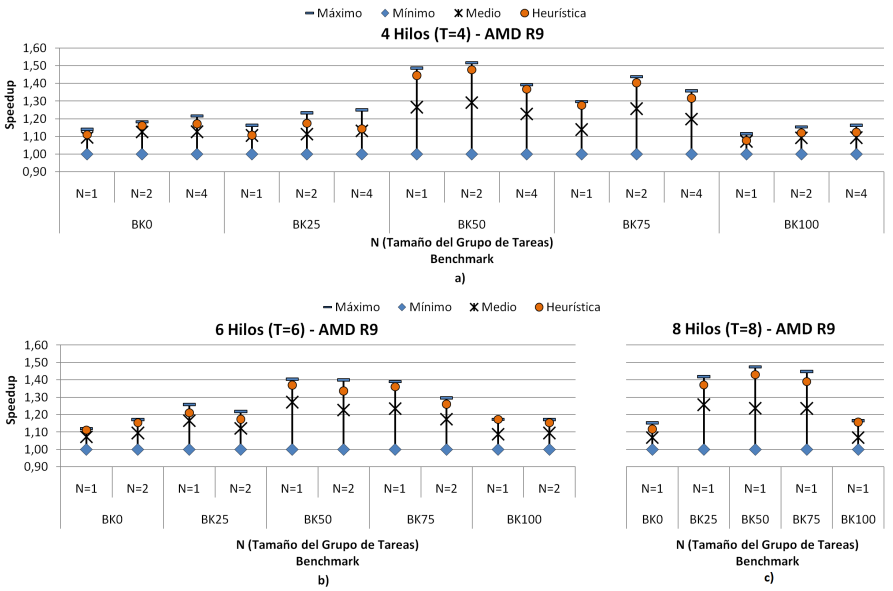


Figura 3.15: Speedups sin CKE conseguidos con tareas reales, para cada benchmark en el acelerador AMD R9. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística.

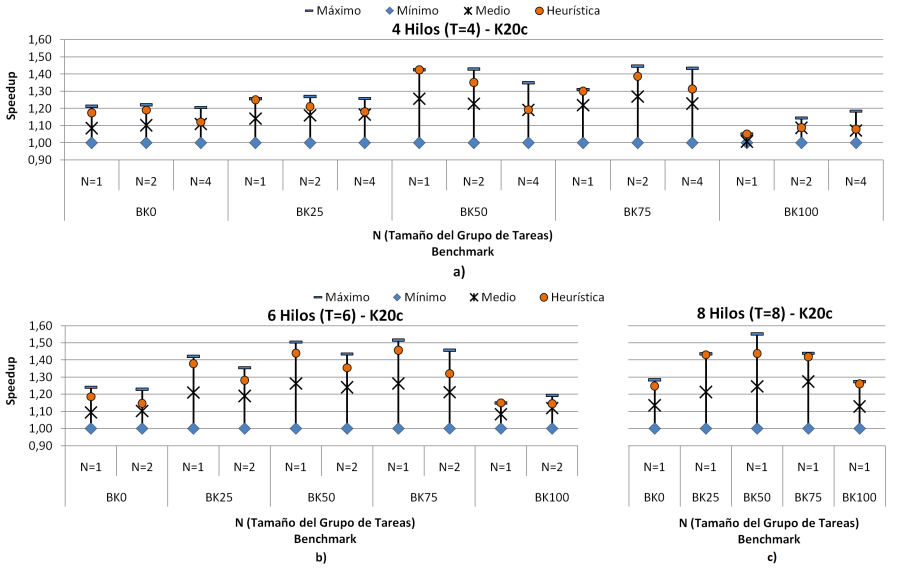


Figura 3.16: Speedups sin CKE conseguidos con tareas reales, para cada benchmark en el acelerador NVIDIA K20c. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística.

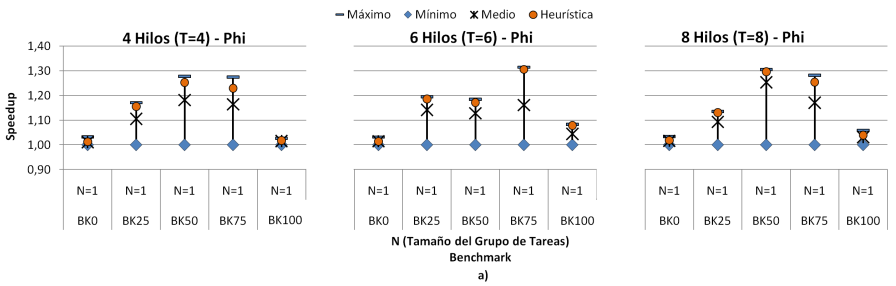


Figura 3.17: Speedups sin CKE conseguidos con tareas reales, para cada benchmark en el acelerador Intel Xeon Phi. Estos speedups son con respecto al tiempo de ejecución de la peor permutación. El máximo speedup se consigue con la mejor permutación, el speedup medio se obtiene usando el tiempo de ejecución medio, y el speedup de la heurística se calcula usando el orden de planificación obtenido por la heurística.

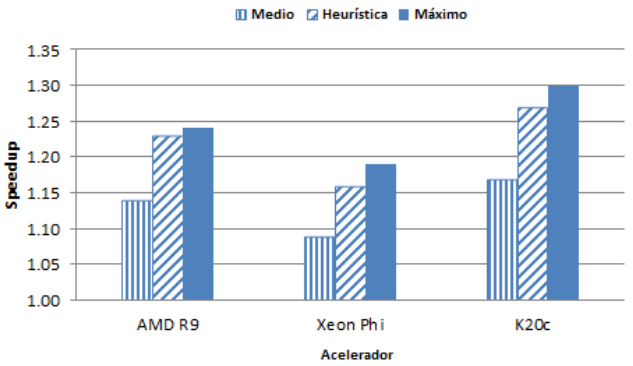


Figura 3.18: Media geométrica del speedup máximo, mínimo y medio para los experimentos de los benchmarks reales.

4 Teoría de Planificación

En este capítulo se aborda el problema de planificar un grupo de tareas desde un punto de vista teórico, haciendo uso de la Teoría de Planificación desarrollada dentro del campo de la Investigación Operativa.

En la Sección 4.1 se resumen algunos conceptos básicos sobre Teoría de la Planificación y como se pueden aplicar al caso particular de ejecución de tareas en un acelerador. Este enfoque distinto permite expresar este caso como un problema de tipo *Flow Shop* para el cual ya existen en la literatura diversas soluciones que serán presentadas en la Sección 4.2, junto con una nueva solución que combina una de las soluciones aportadas por la Teoría de Planificación con nuestro modelo de ejecución de tareas presentado en la Sección 2.4. La validez de esta nueva solución es analizada con una serie de experimentos en la Sección 4.3.

4.1. Teoría de Planificación

El problema de ejecutar N tareas en un acelerador se puede estudiar mediante la teoría de planificación. En [38], Lázaro et al. aplican la teoría de planificación y el modelo de ejecución de tareas propuesto en [40], para diseñar una nueva heurística de planificación de tareas en GPU. La planificación es un proceso de toma de decisión donde n trabajos o tareas deben ser asignadas a m máquinas. Siguiendo la notación introducida por Graham et al. [28], el problema se puede definir por medio de tres campos, α (que describe el entorno de las máquinas), β (las características y restricciones del procesamiento) y γ (la función objetivo).

El lanzamiento de una tarea (trabajo en la terminología de planificación) en un acelerador implica ejecutar al menos un comando de los siguientes tipos: una

transferencia desde el *host* al acelerador (*HtD*), un comando *kernel* (*K*) y una transferencia desde el acelerador hacia el *host* (*DtH*). Una tarea puede estar compuesta por uno, varios o ningún comando de cada uno de los tipos anteriores. Las transferencias de memoria se ejecutan en los motores DMA, mientras que los comandos de *kernel* son ejecutados por el acelerador. Por una parte, las arquitecturas actuales permiten la ejecución concurrente de *kernels* (CKE) si estos comandos no saturan los recursos hardware (memoria, registros, etc) pero, en aplicaciones reales, la mayoría de los *kernels* están diseñados para utilizar completamente todos los recursos, por lo que nuestro análisis se limitará al estudio de *kernels* que no se ejecutan concurrentemente. Por otra parte, los aceleradores modernos disponen de dos motores DMA que permiten que se puedan ejecutar al mismo tiempo dos transferencias en direcciones opuestas. De esta manera, podemos considerar 3 máquinas en nuestro entorno (dos motores DMA además del acelerador), lo cual se corresponde en teoría de planificación a un problema denominado *Flow Shop*, en el que cada trabajo se procesa en cada máquina en un orden determinado. En este caso, el campo α se representa como $F3$ para describir un problema *Flow Shop* compuesto por 3 máquinas. La Figura 4.1 muestra un ejemplo con dos trabajos para un problema *Flow Shop* compuesto por 3 máquinas.

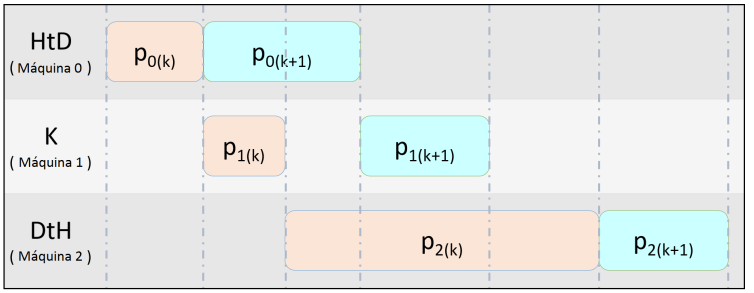


Figura 4.1: Representación del lanzamiento de tareas en un acelerador como un problema *Flow Shop* de 3 máquinas. Los comandos *HtD* se ejecutan en la máquina 0, los comandos *K* en la máquina 1 y los comandos *DtH* en la máquina 2. Los comandos del trabajo k preceden a los comandos del trabajo $(k+1)$ en todas las máquinas. $p_{i(j)}$ corresponde al tiempo de procesamiento del trabajo j en la máquina i.

Una generalización de este problema se denomina *Flexible Flow Shop* (también conocido como *Hybrid Flow Shop*), donde en lugar de m máquinas en serie, hay c etapas en serie. Cada etapa esta constituida por un número de máquinas idénticas en paralelo y se puede usar para describir un clúster de aceleradores. De esta

manera, habría 3 etapas (una etapa para cada tipo de comando), con tantas máquinas como aceleradores tenga el clúster, y el problema se representaría como $FF3$.

Considerando los requisitos y características del procesamiento, se pueden considerar varias posibilidades. Por ejemplo, en un problema *Flow Shop* típico se considera que en todas las máquinas se implementa una política denominada *First Come First Served* (FCFS). Esta política no es válida para aceleradores con Hyper-Q a menos que se utilice algún mecanismo de sincronización para forzar un orden. Por otra parte, se puede demostrar que para los sistemas $F3$ siempre hay planificaciones óptimas que no requieren ningún cambio entre máquinas [63] y, en esos casos, se pueden realizar algunas simplificaciones para hacer más fácil la búsqueda de una planificación óptima. Precisamente en este trabajo se utilizan eventos o una sola cola hardware de Hyper-Q para forzar el mismo orden en todas las máquinas así que de forma directa se puede implementar la política FCFS usando nuestro sistema. Este tipo de problemas se denomina *Permutation Flow Shop* y se identifica usando la palabra *prmu* en el campo β. En el ejemplo mostrado en la Figura 4.1 se puede ver que los comandos del trabajo k preceden a los comandos del trabajo $k+1$ en todas las máquinas.

En un escenario real, uno o más procesos (o hilos) ejecutan parte de su computación en la CPU o *host* y otra parte en el acelerador. Por tanto, las tareas planificadas pueden llegar al sistema en un tiempo particular denominado instante de lanzamiento (r_j). Estas tareas no se pueden lanzar antes de su instante de lanzamiento y esto se indica añadiendo el valor r_j, al campo β. Además, cada proceso puede ejecutar varias tareas en el acelerador, con algunos requisitos entre ellas para asegurar su correcta ejecución. Estos requisitos o dependencias son normalmente codificados mediante un grafo acíclico y se puede expresar usando la palabra *prec* en el campo β. Finalmente, las tareas pueden pertenecer a diferentes usuarios y se deben lanzar en diferentes contextos del acelerador. En la teoría de planificación esto es similar a las familias de trabajos, es decir, trabajos de la misma familia se pueden procesar uno después del otro sin ningún retraso, pero el intercambio de una familia a otra requiere de un tiempo de preparación del contexto de la misma, s, antes de que se lancen los trabajos. Este tiempo de preparación es el tiempo que se consume en la creación del contexto del acelerador, y en la teoría de planificación se expresa añadiendo la palabra $fmls$ al campo β.

Además, es posible también incluir información de las características del trabajo en el campo β, como su tiempo de procesamiento o su prioridad. Por ejemplo, nos referiremos a p_{ij} como el tiempo de procesamiento del trabajo j en la máquina i (como en la Figura 4.1, donde $p_{0(k)}$ es el tiempo de procesamiento del trabajo

Campo	Valor	Significado	Descripción
α	Fm	Flow Shop con m máquinas	Cada trabajo es procesado en un orden dado.
	FFc	Flexible (or Hybrid) Flow Shop con c etapas	Como Fm pero usando c etapas de máquinas idénticas
β	$prmu$	Permutación	Las máquinas usan una política FIFO
	r_j	Tiempo de Lanzamiento	Las tareas llegan en un determinado tiempo
	w_j	factores de Prioridad	A las tareas se asigna un factor de peso o prioridad
	d_j	Tiempos de Finalización	A las tareas se asigna un tiempo de finalización
	$prec$	Dependencias	Las tareas deben seguir un grafo acíclico dirigido
	$fmls$	Familias de Trabajos	Las tareas pertenecen a diferentes usuarios
γ	C_{max}	Makespan	Minimiza el tiempo de finalización de la última tarea
	$\sum w_j C_j$	Tiempo total sopesado de finalización	Incluye factores de prioridad

Tabla 4.1: Notación de la Teoría de Planificación

k en la máquina 0 y así en adelante). Aunque es posible obtener de antemano una estimación de los tiempos de procesamiento de cada comando, los tiempos de transferencias reales dependen de si hay otra transferencia en la dirección opuesta [40]. Por tanto, estas fuentes de incertidumbres pueden llevar a planificaciones no cercanas a la óptima, cuando resolvemos la función objetivo. Se puede asignar también a cada trabajo un factor de prioridad, usando un valor de peso, w_j, que puede ser considerado por la función objetivo.

En último lugar, se debe considerar la función objetivo que define la planificación óptima. Si el tiempo de finalización del trabajo j se indica mediante C_j, la función más común en problemas *Flow Shop* consiste en minimizar el *makespan* C_{max} definido como máx$(C_1, ..., C_n)$, y es equivalente a minimizar el tiempo de finalización del último trabajo en abandonar el 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 de finalización ponderado ($\sum w_j C_j$), también denominado como tiempo de flujo ponderado, que tiene en cuenta los factores de prioridad dados por los valores de ponderación.

La Tabla 4.1 resumen estos valores junto con su significado y una breve descripción. Todos los parámetros se pueden combinar para expresar diferentes situaciones del mundo real y seleccionar una solución adecuada de la literatura existente [22, 3]. El único requisito es obtener la notación correcta que captura el problema. Por ejemplo, un escenario simple con un solo acelerador y varias tareas independientes, listas para ser ejecutadas usando el mismo contexto, se puede modelar como un problema $F3|prmu|Cmax$. Por otra parte, un escenario más complejo como un sistema multi-GPU, donde varios usuarios lanzan tareas independientes en diferentes instantes de lanzamiento, se puede estudiar como un problema $FF3|r_j, fmls|Cmax$. A modo de ejemplo práctico, en la siguiente sección estudiaremos el problema $F3|prmu|Cmax$ que surge cuando varios hilos CPU lanzan tareas independientes que pueden ser procesadas en el mismo contexto del acelerador.

4.2. Búsqueda de una Planificación Óptima

El problema que se presenta en esta sección es típico en escenarios donde uno o más hilos CPU pueden lanzar varias tareas que se pueden ejecutar usando el mismo contexto del acelerador. Por ejemplo, en una aplicación de vídeo vigilancia, podrían haber varios hilos CPU analizando diferentes flujos de vídeo y delegando parte de la computación al acelerador. De una manera similar a CUDA MultiProcess-Service (MPS [57]), un hilo proxy podría ser el encargado de recolectar todas las tareas y ejecutarlas usando el mimo acelerador para mejorar su concurrencia. Este sistema se correspondería con el descrito en la Sección 3.2.

Como cada tarea puede estar compuesta por varios comandos (por ejemplo, comandos *HtD*, *K* y *DtH*) que son ejecutados mediante los motores DMA y el acelerador, entonces el tiempo total de ejecución (el *makespan* C_{max}) se puede reducir seleccionando una planificación de las tareas adecuada. Más concretamente, y siguiendo la notación de la sección anterior, este problema se puede modelar como un problema $F3|prmu|Cmax$, donde estos comandos (*trabajos*) son planificados en tres máquinas.

En un caso más general, la aplicación que ejecuta cada hilo CPU necesitará ejecutar varias tareas en orden. El hilo proxy lanza estas tareas asíncronamente, por lo que se debe asegurar que se mantengan sus dependencias mediante la utilización de eventos o lanzándolas por la misma cola de comandos. Independientemente de ello, la primera tarea de cada hilo CPU podría estar disponible para su ejecución por el hilo proxy en un tiempo dado y el problema sigue pudiéndose modelar como un problema $F3|prmu|Cmax$, considerando solamente la

primera tarea de cada hilo. Adicionalmente, si se conoce de antemano el grafo de dependencias de tareas de cada hilo, entonces se puede formular como un problema $F3|prmu, prec|Cmax$ y se podría obtener una solución de planificación más cercana a la óptima.

Se ha demostrado que el problema $F3||Cmax$ es un problema del tipo *NP-hard* [23]. De hecho, para un caso simple con N tareas independientes habría $N!$ permutaciones diferentes, por lo que se han presentado muchas heurísticas para resolver eficientemente este problema [66]. Estas heurísticas se pueden clasificar como *constructivas* o *de mejora* dependiendo de si realizan la planificación desde cero o mejoran una solución previa. En esta tesis estamos interesados en la planificación en tiempo real, por lo que consideraremos algoritmos con una baja carga computacional, y que hayan obtenido buenos resultados para otros problemas de tipo *Flow Shop*.

Aparte de las heurísticas que se describen en las secciones siguientes, se ha considerado otra heurística que resuelve el problema usando programación MIP (*Mixed Integer Programming*) [84]. Para este propósito se definen varias variables, como el tiempo de inactividad en las máquinas entre trabajos o el tiempo de espera de trabajos entre máquinas, y una función objetivo que minimiza el tiempo de inactividad en la última máquina. La optimización de esta función consiste en la exploración de un árbol de soluciones y, aunque se utilicen técnicas avanzadas de búsqueda, el coste computacional es demasiado elevado para un sistema de tiempo real. Esta técnica se implementó en Matlab, a modo de referencia, pero sus resultados experimentales son inferiores a los obtenidos con las mejores heurísticas que se explican a continuación por lo que fue finalmente descartada.

En los apartados siguientes se detallan los algoritmos seleccionados y como se han adaptado al problema de encontrar una planificación adecuada para un grupo de tareas que se quieren ejecutar en un acelerador.

4.2.1. Heurística de la Pendiente

El problema F2$||C_{max}$ fue uno de los primeros en ser analizados por S.M. Johnson [32]. Con la regla de Johnson se puede encontrar fácilmente una planificación óptima que minimice C_{max} para el caso de 2 máquinas. En primer lugar, se construyen dos conjuntos: el conjunto I contiene todos los trabajos que cumplen la condición $p_{1j} < p_{2j}$, y el conjunto II está compuesto por los trabajos restantes. Los trabajos en el conjunto I van primero en orden creciente de p_{1j}, y los trabajos en el conjunto II van en orden decreciente de p_{2j}. Esta planificación es óptima para el problema $F2||C_{max}$ y se puede extender a problemas $F3$ considerando

que el conjunto I contiene los trabajos que cumplan $p_{1j} + p_{2j} < p_{2j} + p_{3j}$, y el conjunto II contiene los trabajos restantes.

Siguiendo una aproximación similar se presentaron otras heurísticas que tienen en cuenta un número arbitrario de máquinas, como la heurística de la pendiente (*Slope heuristic*) [61], para encontrar planificaciones cercanas a la óptima. Esta heurística da prioridad a las tareas que tienen una mayor tendencia a progresar desde tiempos pequeños a tiempos mayores en la secuencia de los procesos. En nuestro problema con un acelerador, esto se traduce en que se prioriza las tareas con tiempos *HtD* pequeños y tiempos de *DtH* grandes. Esta prioridad se establece mediante un índice de pendiente A_j para cada trabajo j como $A_j = -\sum_{i=1}^{m}(m - (2i-1))p_{ij}$ donde m es el número de máquinas y los trabajos se ordenan en orden decreciente de este índice. Nos referiremos a esta heurística con las siglas SI (*Slope Index*).

4.2.2. Heurística de Planificación NEH

Una de las mejores heurísticas de planificación para problemas *Flow Shop* fue propuesta por Nawaz, Enscore y Ham en [51]. El algoritmo, denominado NEH por sus autores, es una heurística constructiva que itera para encontrar una solución:

1. Para cada tarea calcula su tiempo total de ejecución y los ordena en orden decreciente.

2. Selecciona las dos primeras tareas y encuentra el mejor orden calculando su *makespan*..

3. Para cada una de las tareas restantes, $i = 3, \ldots, n$, encuentra la mejor planificación, colocándolas en todas las posibles posiciones i en la secuencia de tareas que ya han sido planificadas y calculando el *makespan* de cada una de ellas.

Uno de los principales inconvenientes de este algoritmo es que el *makespan* se debe calcular $[n(n+1)/2] - 1$ veces, de las cuales n son secuencias completas y el resto son planificaciones parciales, pero puede consistentemente obtener mejores resultados que otras heurísticas [75]. Otro inconveniente es que el cálculo del *makespan* no tiene en cuenta los efectos del solapamiento de transferencias, por lo que los resultados no son tan buenos como en los problemas clásicos de tipo *Flow Shop*.

4.2.3. Heurística de Planificación NEH-GPU

Basándonos en el modelo de ejecución de [40] y en la heurística que acabamos de ver (NEH, [51]) hemos desarrollado un nuevo algoritmo que intenta aprovechar las ventajas del marco teórico de la teoría de planificación y del modelo de ejecución de grupos de tareas en una GPU.

En la heurística NEH, el *makespan* se calcula usando una duración fija para cada comando, pero en una GPU la duración de dos comandos de transferencia solapados puede variar más de un 10 %. Por tanto, la predicción del *makespan* puede tener un gran error que puede llevar a una selección errónea en el orden de lanzamiento de las tareas. En esta tesis proponemos combinar el método NEH con un modelo de ejecución de tareas en GPU que puede predecir con más exactitud el *makespan*. En lugar de usar alguno de los algoritmos de cálculo del *makespan* propuestos en [51] o en [75], hemos usado el modelo de ejecución desarrollado en esta tesis para crear esta nueva heurística a la que denominaremos NEH-GPU a partir de ahora.

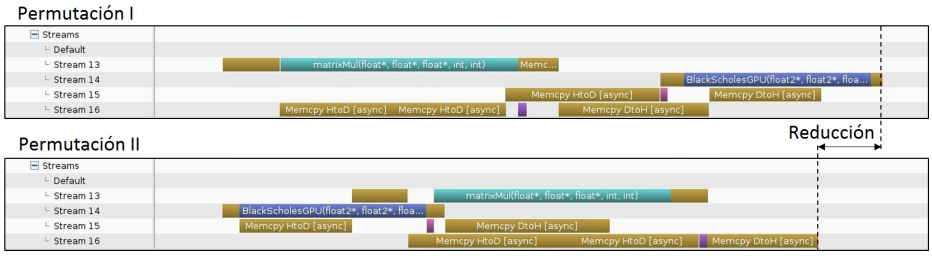


Figura 4.2: Ejemplo del lanzamiento de 4 tareas independientes en una GPU NVIDIA K20c. Las tareas *Matrix Multiplication*, *Black Scholes*, *Matrix Transposition* y *Vector Addition* se lanzan por los streams 13, 14, 15 y 16 respectivamente. La imagen correspondiente a la permutación I muestra el orden seleccionado por una heurística de planificación, mientras que la imagen de la permutación II corresponde al orden seleccionado por otra heurística. La reducción del *makespan* se debe únicamente a los diferentes órdenes de lanzamiento de tareas.

Para ilustrar las ventajas de usar el modelo de ejecución discutiremos el ejemplo mostrado en la Figura 4.2. Los cronogramas de la figura comparan los resultados que se obtienen mediante la heurística NEH (Permutación I) con los obtenidos con la heurística NEH-GPU (Permutación II) cuando se planifican cuatro tareas independientes: *Matrix Multiplication* (MM), *Black-Scholes* (BS), *Matrix Transposition* (TM) y *Vector Addition* (VA). Para cada tarea se dan, en la Tabla 4.2,

ID Stream	Tarea	HtD (ms)	K (ms)	DtH (ms)	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 4.2: Tiempo de ejecución en milisegundos de los comandos de las tareas *Matrix Multiplication* (MM), *Black-Scholes* (BS), *Matrix Transposition* (TM) y *Vector Addition* (VA) usadas en el ejemplo de la Figura 4.2. La columna *Total* muestra el *makespan* de la tarea cuando se ejecuta sola.

Iteración	NEH		NEH-GPU	
	Orden	Makespan (ms)	Orden	Makespan (ms)
1º	3-0	24.73	3-0	25.32
	0-3	**19.37**	0-3	**19.37**
2º	2-0-3	24.40	2-0-3	**25.57**
	0-2-3	24.35	0-2-3	26.26
	0-3-2	**24.35**	0-3-2	26.26
3º	1-0-3-2	31.06	1-2-0-3	**26.58**
	0-1-3-2	32.08	2-1-0-3	31.28
	0-3-1-2	27.47	2-0-1-3	33.04
	0-3-2-1	**27.32**	2-0-3-1	29.30

Tabla 4.3: *Makespan* parciales obtenidos por las heurísticas NEH y NEH-GPU al lanzar las tareas *Matrix Multiplication* (MM), *Black-Scholes* (BS), *Matrix Transposition* (TM) y *Vector Addition* (VA) usadas en el ejemplo de la Figura 4.2 del ejemplo de la Figura 4.2.

la duración de sus comandos *HtD*, *K* y *DtH*, y sus *makespan* totales si las tareas son lanzadas solas. Además, en la Tabla 4.3 se muestran los *makespan* parciales obtenidos por las heurísticas NEH y NEH-GPU. Las dos tareas más largas, 3 y 0, se lanzan en la primera iteración. Cada algoritmo calcula el *makespan* usando los datos de la Tabla 4.2. Aunque ambos predicen un *makespan* diferente para el orden $3-0$, seleccionan el mismo orden $0-3$, ya que es el que tiene mejor *makespan*. En la segunda iteración intentan todas las permutaciones posibles insertando la tarea 2 en el orden $0-3$ y predicen *makespan* diferentes, lo que les lleva a seleccionar diferentes órdenes de lanzamiento $0-3-2$ para NEH y $2-0-3$ para NEH-GPU. Por último, en la última iteración, cada heurística inserta la última tarea por insertar para obtener diferentes órdenes de planificación. La heurística NEH predice un *makespan* de 27.32 ms cuando utiliza el orden de planificación $0-3-2-1$, aunque la ejecución real usando este orden es de 29.79 ms ya que no ha tenido en cuenta el efecto del solapamiento al predecir el *makespan*. Por otro lado, la heurística NEH-GPU predice un *makespan* de 26.58 ms usando el orden $1-0-3-2$ y la ejecución real toma un valor de 26.97 ms. Esto significa que el modelo de ejecución de tareas en GPU predice de una forma más exacta el tiempo de ejecución real, y gracias a ello puede tomar una decisión mejor de planificación.

4.3. Validación

En esta sección se presentan los experimentos que se han llevado a cabo para evaluar los tres algoritmos de la Sección 4.2 (SI, NEH y NEH-GPU) más la heurística *Single Queue* presentada en la Sección 3.4, en un escenario multihilo (por ejemplo, un servidor de computación heterogénea) y sobre un entorno NVIDIA-CUDA. El objetivo de la evaluación es, en primer lugar, comprobar si la heurística *Single Queue*, que fue desarrollada de forma específica para la ejecución en aceleradores, sigue siendo tan eficiente al compararla con algoritmos con una base teórica y, en segundo lugar, determinar si el algoritmo NEH-GPU, que se ha desarrollado combinando la teoría de planificación con nuestro modelo de ejecución, es capaz de mejorar los resultados obtenidos con cualquier otro método.

Siguiendo el esquema de la Sección 3.5, los experimentos se realizan en un escenario donde varios hilos envían una o varias tareas a un acelerador para su ejecución. Los hilos envían la información de las tareas mediante funciones CUDA a un buffer que es constantemente consultado por un hilo proxy CPU. Este hilo es el encargado de reordenar el conjunto de tareas encontrado en el buffer,

usando una de las cuatro heurísticas, y enviar los comandos correspondientes al acelerador. Para los experimentos hemos considerado tres escenarios con 4, 8 y 16 hilos para reflejar distintas cargas de trabajo.

Todos los experimentos han sido llevados a cabo usando el conjunto de tareas reales presentadas en la Tabla 3.2. Este conjunto de tareas con distintos tiempos de computación y de transferencias se ha obtenido de los SDK de CUDA y de Rodinia como en [40]. Estas tareas se pueden clasificar como de transferencias dominantes (DT) o de kernel dominante (DK), dependiendo de si el tiempo de transferencia domina sobre el tiempo de computación o viceversa. Aunque algunas de estas tareas tienen varios comandos del mismo tipo (por ejemplo, la tarea *MatrixMult* tiene que transferir dos matrices desde el *host* al *device*, o lanzar varios comandos de computación como es el caso de la tarea *Pathfinder*), el planificador considera un solo comando que encapsula a todos los comandos del mismo tipo.

Para representar una carga computacional más variable, se han seleccionado diferentes parámetros de entrada para incrementar el número de tareas hasta 21. El conjunto de tareas utilizado se muestra en las Tablas 4.4, 4.5 y 4.6. Además, para incrementar la relevancia de los resultados, se han utilizado tres arquitecturas de NVIDIA diferentes, K20c (Kepler), GTX980 (Maxwell) y Titan X (Pascal). Por último, cada experimento ha sido ejecutado 15 veces para obtener la media de su *makespan*.

4.3.1. Análisis Estadístico

En primer lugar, analizaremos estadísticamente nuestro modelo de ejecución de tareas concurrentes, para demostrar la validez de la nueva heurística y compararla con el resto de heurísticas. Para este fin, se han obtenido todas las combinaciones posibles de 4 tareas, con repetición, del conjunto de 21 tareas presentado anteriormente en las Tablas 4.4, 4.5 y 4.6. Esto representa un total de $\left(\binom{21}{4}\right) = 10626$ benchmarks de 4 tareas cada uno. Para calcular esta cantidad se ha usado el coeficiente binomial $\left(\binom{n}{k}\right)$, el cual permite indicar como seleccionar k objetos de un total de n, permitiendo que la selección se pueda repetir y cuya fórmula es $\frac{(n+k-1)!}{k!(n-1)!}$. Para cada benchmark hay $4! = 24$ planificaciones diferentes, por tanto hay 255024 experimentos diferentes que se han ejecutado 15 veces. La Figura 4.3 muestra tres histogramas que recogen el porcentaje de error cometido por el modelo comparado con la ejecución real en las GPUs K20c (arriba), GTX 980 (centro) y Titan X (abajo). Como se puede observar en la Figura 4.3 la mediana de los porcentajes de error del modelo en las tres GPUs está por debajo

GPU	Kernel	HtD (ms)	K (ms)	DtH (ms)	oHtD (ms)	oDtH (ms)	Parámetros					
K20c	MM	2.522	10.605	1.262	3.565	1.804	0	4	1024	1024	1024	1024
	MM	5.028	21.185	1.260	7.126	1.792	0	4	2048	1024	1024	2048
	MM	5.028	10.732	0.328	7.112	0.464	0	4	4096	512	512	4096
	BS	0.733	8.499	0.491	1.033	0.694	2	2	200000	24000		
	CONV	10.041	1.680	10.017	14.190	14.321	4	3	256	32768	1	
	CONV	0.330	23.231	0.327	0.461	0.460	4	3	512	512	300	
	CONV	5.028	0.853	5.017	7.111	7.128	4	3	512	8192	1	
	CONV	10.040	1.686	10.017	14.205	14.239	4	3	512	16384	1	
	CONV	1.269	23.305	1.260	1.794	1.792	4	3	1024	1024	100	
	CONV	1.269	4.635	1.262	1.792	1.798	4	3	1024	1024	20	
	CONV	5.028	12.898	4.991	7.126	7.120	4	3	2048	2048	15	
	TM	5.028	0.312	4.998	7.113	7.165	5	2	2048	2048		
	TM	10.041	0.634	10.017	14.190	14.321	5	2	2048	4096		
	TM	10.041	0.617	10.017	14.190	14.321	5	2	4096	2048		
	FWT	2.523	1.377	1.261	3.569	1.796	6	2	20	4		
	FWT	5.028	2.680	2.507	7.117	3.579	6	2	21	5		
	FWT	10.040	5.955	4.998	14.210	7.165	6	2	22	6		
	PAF	0.203	5.267	0.078	0.281	0.105	8	4	128	128	2	13000
	PF	3.601	3.966	0.017	5.093	0.017	9	3	500	6000	20	
	VA	10.040	0.366	4.998	14.210	7.165	10	1	4194304			
	VA	12.547	0.452	6.267	17.733	8.957	10	1	5242880			

Tabla 4.4: Selección de parámetros de entrada, de las tareas de la Tabla 3.2, para la GPU K20c. La tabla muestra para cada tarea los tiempos de ejecución en milisegundos de los comandos de transferencia no solapados (HtD y DtH) y solapados (oHtD y oDtH), así como también, del tiempo de los comandos *kernel* (K).

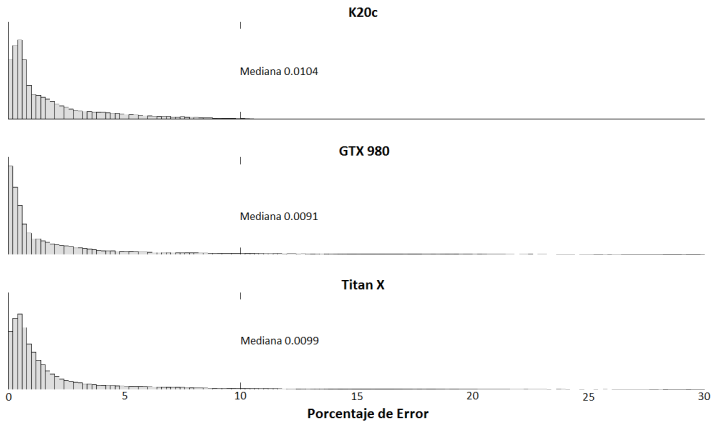


Figura 4.3: Porcentaje de error cometido por el modelo comparado con la ejecución real de los en las GPUs K20c (arriba), GTX 980 (centro) y Titan X (abajo)

GPU	Kernel	HtD (ms)	K (ms)	DtH (ms)	oHtD (ms)	oDtH (ms)	Parámetros					
	MM	2.728	12.405	2.796	4.167	4.174	0	4	1024	2048	2048	1024
	MM	5.441	24.653	2.795	8.336	4.169	0	4	2048	2048	2048	2048
	MM	5.440	11.619	0.709	8.309	1.052	0	4	4096	1024	1024	4096
	BS	1.567	7.119	1.074	2.390	1.600	2	2	800000	96000		
	CONV	10.867	1.603	11.146	16.615	16.684	4	3	1024	16384	1	
	CONV	0.692	17.423	0.709	1.051	1.052	4	3	1024	1024	150	
	CONV	5.440	0.808	5.579	8.309	8.324	4	3	1024	8192	1	
	CONV	11.546	1.702	11.841	17.646	17.665	4	3	1024	17408	1	
GTX	CONV	2.728	20.743	2.795	4.165	4.174	4	3	2048	2048	50	
980	CONV	2.728	6.187	2.795	4.175	4.169	4	3	2048	2048	15	
	CONV	2.727	10.344	2.795	4.169	4.174	4	3	2048	2048	25	
	TM	5.441	0.419	5.579	8.334	8.324	5	2	4096	2048		
	TM	10.867	0.875	11.146	16.615	16.684	5	2	2048	8192		
	TM	10.867	0.831	11.144	16.651	16.628	5	2	4096	4096		
	FWT	2.727	2.091	1.404	4.173	2.090	6	2	21	5		
	FWT	5.441	4.652	2.796	8.311	4.166	6	2	22	6		
	FWT	10.868	9.225	5.579	16.609	8.320	6	2	23	7		
	PAF	0.165	5.001	0.064	0.246	0.090	8	4	512	512	3	19500
	PF	4.381	5.953	0.013	6.710	0.013	9	3	750	9000	1	
	VA	10.868	0.596	5.578	16.635	8.300	10	1	8388608			
	VA	13.580	0.741	6.970	20.775	10.413	10	1	10485760			

Tabla 4.5: Selección de parámetros de entrada, de las tareas de la Tabla 3.2, para la GPU GTX 980. La tabla muestra para cada tarea los tiempos de ejecución en milisegundos de los comandos de transferencia no solapados (HtD y DtH) y solapados (oHtD y oDtH), así como también, el tiempo de los comandos *kernel* (K).

GPU	Kernel	HtD (ms)	K (ms)	DtH (ms)	oHtD (ms)	oDtH (ms)	Parámetros					
Titan X	MM	2.879	10.724	1.296	3.122	1.507	0	4	2048	2048	2048	2048
	MM	2.813	21.039	5.100	3.124	5.936	0	4	1024	4096	4096	1024
	MM	4.925	9.183	0.327	5.467	0.379	0	4	7168	1024	1024	7168
	BS	0.819	9.009	0.495	0.891	0.570	2	2	800000	96000		
	CONV	11.323	1.508	10.191	12.435	11.866	4	3	2048	16384	1	
	CONV	1.412	15.838	1.282	1.567	1.491	4	3	2048	2048	80	
	CONV	5.644	0.760	5.286	6.333	5.941	4	3	2048	8192	1	
	CONV	15.348	1.602	18.396	33.050	32.779	4	3	2048	17408	1	
	CONV	1.815	18.733	2.173	3.897	3.865	4	3	2048	2048	95	
	CONV	1.815	8.859	2.173	3.897	3.865	4	3	2048	2048	45	
	CONV	1.815	11.820	2.173	3.897	3.865	4	3	2048	2048	60	
	TM	6.789	0.474	8.092	14.580	14.448	5	2	5120	3072		
	TM	16.279	1.049	19.408	34.977	34.662	5	2	4096	9216		
	TM	14.471	0.912	17.253	31.091	30.812	5	2	4096	8192		
	FWT	3.626	2.253	2.165	7.781	3.860	6	2	22	6		
	FWT	7.238	4.481	4.321	15.551	7.721	6	2	23	7		
	FWT	14.466	9.961	8.633	31.091	15.433	6	2	24	8		
	PAF	0.178	6.372	0.077	0.372	0.129	8	4	512	512	5	32500
	PF	5.180	5.042	0.010	11.126	0.011	9	3	1000	12000	1	
	VA	10.492	0.526	4.782	11.675	5.575	10	1	15728640			
	VA	12.205	0.593	5.415	13.226	6.333	10	1	17825792			

Tabla 4.6: Selección de parámetros de entrada, de las tareas de la Tabla 3.2, para la GPU Titan X. La tabla muestra para cada tarea los tiempos de ejecución en milisegundos de los comandos de transferencia no solapados (HtD y DtH) y solapados (oHtD y oDtH), así como también, el tiempo de los comandos *kernel* (K).

GPU	SI	NEH	SQ	NEH-GPU	BEST
K20c	1.024	1.064	1.038	1.066	1.081
GTX 980	1.001	1.041	1.037	1.063	1.089
Titan X	1.031	1.090	1.061	1.091	1.097

Tabla 4.7: Speed-up de la mediana de cada una de las heurísticas (SI, NEH, SQ y NEH-GPU) y para el mejor *makespan* (BEST) en las tarjetas K20c, GTX 980 y Titan X.

del 0.02 %, confirmando los resultados que se obtuvieron en la Sección 2.6.

Además, se ha obtenido una media del *makespan* para cada uno de los 255024 experimentos. Hemos seleccionado el *makespan* mínimo y medio para cada benchmark y también el valor medio del *makespan* para cada heurística. Por último, hemos obtenido el valor de *speed-up* de cada heurística con respecto al valor medio por benchmark, y también el *speed-up* del tiempo mínimo de ejecución con respecto a la media (el mejor valor de *speed-up*). Estos valores se representan en la Figura 4.4 para las GPUs K20c (arriba), GTX980 (centro) y Titan X (abajo) usando el método *box plot* para mostrar sus propiedades estadísticas [78]. Los diagramas de caja o *box plot* son especialmente útiles cuando la distribución de una variable es asimétrica o se aleja de la distribución normal. En este tipo de casos interpretar una variable en función de su media o desviación estándar es un error puesto que estos estimadores no describen fielmente las características de la muestra. En algunas ocasiones, cuando la muestra es lo suficientemente grande, se dibujan unas muescas dentro de los cuadrados que representan los intervalos de confianza al 95 % para la mediana. Con estas muescas podemos evaluar si las diferencias entre dos variables son significativas si las muescas no se superponen entre ambas variables. En cada caja, la línea horizontal central corresponde a la mediana, los ejes de la caja corresponden el segundo y tercer cuartil. Los bigotes se dibujan con líneas discontinuas y se extienden hasta la mayoría de los puntos de datos más extremos, que no se consideran como valores atípicos (sobre el $\pm 2{,}7\sigma$ y 99,3 % de cobertura). Los valores atípicos son representados con un símbolo '+' y hay muescas sobre las marcas de la mediana para los intervalos de confianza. Dos medianas son significativamente diferentes en un 5 % si sus intervalos no se solapan.

La columna de cajas más a la izquierda en la Figura 4.4 corresponde a la heurística de la pendiente SI. Esta heurística es muy simple pero sus resultados no son muy buenos. Por tanto, el valor de su mediana es muy cercano a 1 (eso quiere decir que casi no se obtiene ningún *speed-up*) y en muchos casos el *speed-up* está por debajo de 1. La siguiente columna de cajas muestra los resultados

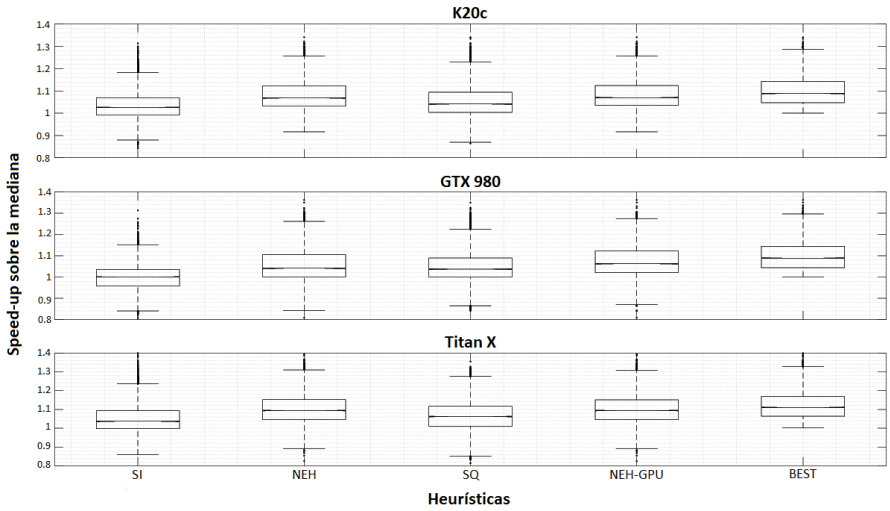


Figura 4.4: Diagramas *box plot* del *speed-up* sobre la mediana del *makespan* de 10626 benchmarks de 4 tareas en las GPUs K20c (arriba), GTX980 (centro) y Titan X (abajo). La marca central corresponde a los valores de la media, las cajas abarcan el segundo y tercer cuartil, los bigotes se extiende hasta cubrir el 99,3 % de los valores, los valores atípicos se muestran individualmente con signos '+' y se usan muescas en las marcas de la media para intervalos de comparación. Se muestran los resultados para cada una de las heurísticas (SI, NEH, SQ y NEH-GPU) y para el mejor *makespan* (BEST).

del algoritmo NEH. Los resultados son mucho mejores, con un *speed-up* medio cercano a 1.05, y la mayoría de los valores de *speed-up* están por encima de 1, aunque los bigotes inferiores se extienden por debajo de 0.9. La columna central de cajas corresponden a la heurística SQ (*Single Queue*). Los resultados son mejor que SI pero no son tan buenos como los de NEH. La mayoría de los valores de *speed-up* están por encima de 1 y los bigotes inferiores se extienden por debajo de 0.9 al igual que en el algoritmo NEH. A continuación, la siguiente columna de cajas hace referencia al nuevo algoritmo NEH-GPU presentado en esta tesis y es el que mejores valores de *speed-up* obtiene. NEH-GPU alcanza resultados muy cercanos al mejor (columna de cajas más a la derecha), con una mediana ligeramente más baja y sus bigotes inferiores se extienden por encima de 0.9. Sus *speed-up* medios son mayores si los comparamos con los obtenidos en las otras heurísticas, y no tiene valores atípicos por debajo de los bigotes inferiores. Las

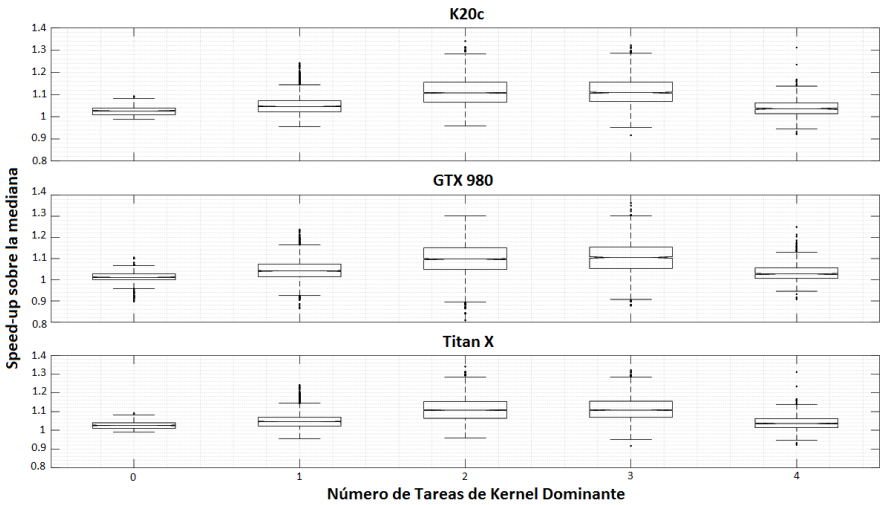


Figura 4.5: Diagramas *box plot* del *speed-up* sobre la mediana del *makespan* de 10626 benchmarks de 4 tareas en las GPUs K20c (arriba), GTX980 (centro) y Titan X (abajo). Se muestran los resultados para la heurística NEH-GPU y diferentes combinaciones de tareas DK y DT (la columna de cajas más a la izquierda solamente considera benchmarks con 4 tareas DT, la siguiente columna de cajas benchmarks con 1 tarea DK y 3 tareas DT y así en adelante.

muescas en todas las cajas están muy juntas pero ninguna de ellas se solapan, por lo que cada heurística es significativamente distinta de las demás desde un punto de vista estadístico. La Tabla 4.7 muestra el *speed-up* para cada una de las heurísticas (SI, NEH, SQ y NEH-GPU) y para el mejor *makespan* (BEST) sobre la mediana en las tarjetas K20c, GTX 980 y Titan X. En la Tabla 4.7 se puede apreciar que la heurística NEH-GPU consigue siempre buenos resultados en las GPUs utilizadas.

Tanto NEH-GPU como los mejores resultados muestran que hay muchos experimentos donde se pueden obtener valores de *speed-up* hasta 1.40, y sería interesante confirmar si los tipos de experimentos que se pueden beneficiar de una buena planificación de tareas son aquellos que tienen una buena mezcla de tareas tal y como ocurría con *Single Queue*. Por tanto, hemos clasificado las tareas como de transferencias dominantes (DT) o de kernel dominante (DK) dependiendo de si el tiempo de transferencia es mayor que el tiempo de computación, y viceversa, y hemos analizado los resultados obtenidos por diferentes combinaciones de tareas DT y DK. La Figura 4.5 muestra los diagramas *box plot* de los valores

de *speed-up* obtenidos por el algoritmo NEH-GPU para las GPUs K20c (arriba), GTX980 (centro) y Titan X (abajo), separando los resultados según la cantidad de tareas DK. La columna de cajas más a la izquierda corresponde a los experimentos donde no existen tareas DK (las cuatro tareas son DT), la siguiente columna de cajas tiene en cuenta los experimentos con 1 tarea DK y 3 tareas DT, y así en adelante. Como se puede observar, los experimentos donde todas las tareas son del mismo tipo obtienen muy poco beneficio de una buena planificación, pero los experimentos donde hay una mezcla equitativa de tareas DK y DT tienen más oportunidades de solapamiento y los valores de *speed-up* pueden ser mucho mayores.

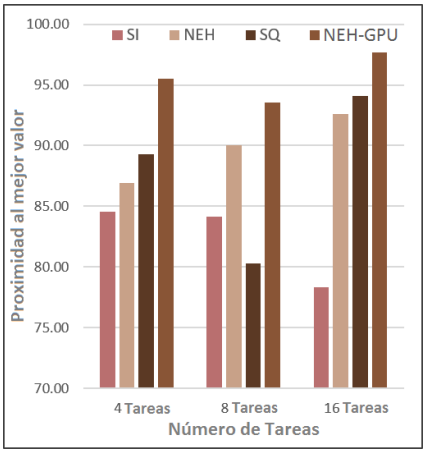


Figura 4.6: Proximidad a los mejores valores con diferentes número de tareas.

4.3.2. Escalabilidad

Al igual que en la sección anterior, en esta sección se evalúan las cuatro heurísticas en un escenario multihilo (por ejemplo, un servidor de computación heterogénea), donde varios hilos ejecutando aplicaciones (*workers*) envían tareas a un acelerador o *device* para su ejecución. Los hilos envían la información de las tareas mediante funciones CUDA, a un buffer que es constantemente consultado por un hilo proxy CPU. Este hilo es el encargado de reordenar el conjunto de tareas encontrado en el buffer, usando una de las cuatro heurísticas, y enviar los comandos correspondientes al acelerador.

En primer lugar se evalúa la escalabilidad de cada heurística incrementando el número de tareas de 4 a 8 y 16. Este número de tareas representa un número

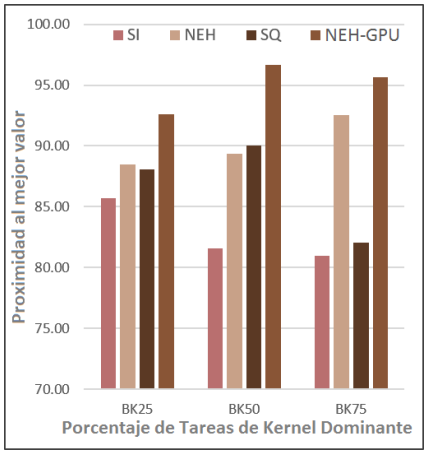


Figura 4.7: Proximidad a los mejores valores con diferente proporción de tareas DK.

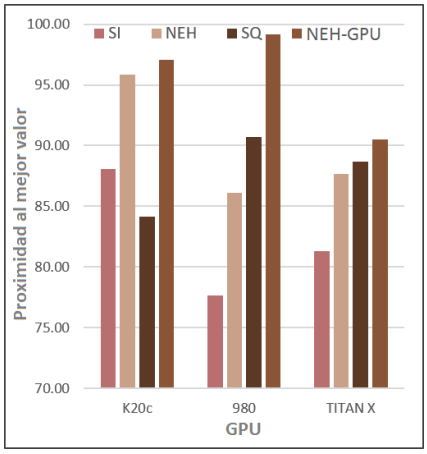


Figura 4.8: Proximidad a los mejores valores con diferentes arquitecturas GPU.

grande de benchmarks diferentes $(\binom{21}{8}) \approx 3 \cdot 10^6$ para 8 tareas y $(\binom{21}{16}) \approx 10^9$ para 16 tareas. Además, hay $8! = 40320$ planificaciones posibles con 8 tareas, y $16! \approx 10^{13}$ con 16 tareas. Por tanto, se ha seleccionado un subconjunto aleatorio de benchmarks diferentes para cada combinación de número de tareas (4, 8, y 16) y características de la GPU (K20c, GTX 980 y Titan X), y se han obtenido solamente los resultados para cada heurística ya que es inviable obtener todas las

permutaciones posibles para 8 o 16 tareas en un tiempo razonable. Para comparar, se ha considerado el mejor *makespan* (BM) para cada benchmark en cada arquitectura, estableciendo así un valor mínimo de referencia. A continuación, para cada experimento, hemos comparado el *makespan* obtenido por cada heurística con este valor de referencia, para confirmar su proximidad. Esta proximidad se calcula dividiendo ambos valores ($BM/HM \cdot 100$), para obtener un porcentaje con valor máximo de 100 % (el mayor y mejor porcentaje). Los resultados incluyen la penalización por la ejecución de la planificación para cada heurística. Esta penalización oscila desde $1\mu s$ para SI y menos de $1ms$ para el resto de heurísticas cuando se planifican 16 tareas. De esta manera, teniendo en cuenta la duración del *makespan*, la penalización por la planificación es insignificante.

Las Figuras 4.6, 4.7 y 4.8 muestran la media que se obtiene con este experimento para cada heurística, según el número de tareas, la proporción de tareas DK, y arquitectura GPU respectivamente. Un valor de proximidad cercano al 100 % significa que la heurística obtiene el mejor *makespan*, o un valor muy cercano, la mayoría de las veces. Se puede ver que la nueva heurística NEH-GPU regularmente obtiene mejores valores de proximidad que las otras heurísticas, en todas las situaciones y arquitecturas GPU.

4.4. Resumen

En este capítulo hemos discutido como aplicar los conceptos de la teoría de la planificación a problemas de planificación de tareas en GPUs. Se ha mostrado que la ejecución concurrente de tareas en GPU usando CUDA streams se puede modelar como un problema *Flow Shop*.

El lanzamiento de una tarea (trabajo en la terminología de planificación) implica ejecutar al menos uno de estos comandos: una transferencia desde el *host* al acelerador, un comando kernel y una transferencia desde el acelerador al *host*. Las transferencias de memoria son ejecutadas por los motores DMA, mientras que los comandos de kernel son ejecutados por el propio acelerador. Las arquitecturas actuales de los aceleradores permiten la ejecución concurrente de kernels (CKE) cuando existen recursos hardware suficientes para que se pueda llevar a cabo la concurrencia. Sin embargo, en aplicaciones reales la mayoría de los comandos kernels saturan los recursos hardware del acelerador, no dejando la opción de que otro comando kernel pueda ejecutarse concurrentemente. Por tanto, nuestro análisis se ha centrado de nuevo en tareas cuyos comandos de kernels saturan todos los recursos del acelerador y no se lleva a cabo CKE.

La ventaja más importante de este enfoque es que se puede definir una función objetivo y se puede seleccionar también una solución apropiada de la literatura existente. Como un ejemplo práctico se ha estudiado el problema $F3|prmu|Cmax$, que surge cuando varios hilos CPU lanzan kernels independientes que pueden ser ejecutados en el mismo contexto GPU. A continuación se ha desarrollado una nueva heurística llamada NEH-GPU, que combina una heurística existente con un modelo de ejecución de tareas en GPU. Esta heurística se puede incluir también como un módulo en tiempo de ejecución ya que no modifica los kernels originales y tiene una penalización insignificante en el tiempo de ejecución.

Se han llevado a cabo varios experimentos que muestran la confianza y robustez de este nuevo enfoque. Para este propósito, se han evaluado tres arquitecturas GPU diferentes (Kepler, Maxwell y Pascal), usando varios kernels reales procedentes de los SDK de CUDA y Rodinia. Se ha llevado a cabo un análisis estadístico para valorar la confianza estadística de la heurística NEH-GPU y mostrar su ventaja sobre otras heurísticas. Además, para valorar la escalabilidad de las heurísticas, se ha variado el número de tareas en los experimentos. En todos ellos, la heurística NEH-GPU ha obtenido los resultados más cercanos al mejor valor de *makespan* obtenido por cualquier heurística.

5 Conclusiones

En esta tesis hemos abordado la planificación de grupos de tareas en aceleradores con el objetivo de disminuir el tiempo total de ejecución y aumentar la ocupación de los recursos hardware. Para ello hemos creado un modelo de ejecución de tareas que permite simular con exactitud los procesos de transferencias de datos y ejecución de *kernels*, y también hemos propuesto varias soluciones al problema de obtener una planificación eficiente de las tareas. Por último, hemos probado tanto el modelo de ejecución como los algoritmos de planificación usando cargas de trabajo reales sobre algunos de los aceleradores más habituales como son las GPUs de NVIDIA y AMD, y los procesadores Xeon Phi de Intel.

En este capítulo se presentan, en la Sección 5.1, las principales contribuciones y conclusiones de esta tesis. A continuación, en la Sección 5.2 se enumeran los artículos publicados con los resultados de esta tesis y, por último, en la Sección 5.3 se discuten algunas posibles líneas futuras de investigación.

5.1. Contribuciones y conclusiones

A lo largo de esta tesis hemos estudiado algunos de los entornos de programación de aceleradores más conocidos como son CUDA y OpenCL. Por ejemplo, los experimentos realizados para CUDA sobre GPUs de NVIDIA muestran que el actual planificador hardware utiliza los requisitos de recursos del *kernel* para seleccionar el orden en que los bloques de hilos son lanzados en la GPU, pero ignora los beneficios de un buen solapamiento entre transferencias de datos y computación. En el Capítulo 1 hemos presentado un ejemplo que muestra el impacto que puede tener sobre el tiempo total de ejecución una mala selección del

orden de ejecución de un grupo de tareas y, es por ello que en esta tesis nos hemos propuesto como objetivo planificar, de forma eficiente y automática, un grupo de tareas para disminuir su tiempo total de ejecución y aumentar la ocupación de los recursos hardware.

En el Capítulo 2 hemos presentado un modelo temporal que simula la ejecución de varias tareas independientes en una GPU soportando la concurrencia entre comandos. El modelo permite caracterizar la ejecución de cada uno de los comandos pertenecientes a una tarea y además simular la interacción de esos comandos con el resto de comandos de otras tareas. También hemos diseñado una serie de benchmarks, compuestos tanto por tareas sintéticas como por tareas reales seleccionadas de varios SDKs, que permiten representar cargas de trabajo de distinta naturaleza. Por último, hemos validado el modelo usando estos benchmarks con algunas arquitecturas actuales de NVIDIA, como Kepler y Maxwell, de AMD, como GCN, o de Intel, como Xeon Phi, obteniendo un error de simulación inferior al 1.5 %.

En el Capítulo 3 hemos presentado un sistema de ejecución de grupos de tareas que permite analizar las tareas que se van a ejecutar en el acelerador y predecir su comportamiento usando el modelo de ejecución que se presentó en el capítulo anterior. Este sistema permite también forzar un orden de ejecución por lo que hemos diseñado una heurística de planificación, capaz de funcionar en tiempo de ejecución, que puede establecer un orden de ejecución de tareas muy cercano al óptimo posible, reduciendo así el tiempo total de ejecución. Además, hemos evaluado el sistema de ejecución y la heurística para determinar su fiabilidad, aplicabilidad y escalabilidad, comparando los valores obtenidos por nuestro sistema con la aplicación de una búsqueda de fuerza bruta del orden óptimo de planificación.

Por último, en el Capítulo 4 hemos discutido como aplicar los conceptos de la teoría de la planificación a problemas de planificación de tareas en GPUs. Se ha mostrado que la ejecución concurrente de tareas en GPU usando CUDA streams se puede modelar como un problema *Flow Shop*. La ventaja más importante de este enfoque es que se puede definir una función objetivo y se puede seleccionar también una solución apropiada de la literatura existente. Como un ejemplo práctico se ha estudiado el problema $F3|prmu|Cmax$, que surge cuando varios hilos lanzan *kernels* independientes que pueden ser ejecutados en el mismo contexto GPU. A continuación, se ha desarrollado una nueva heurística llamada NEH-GPU que combina una heurística existente con un modelo de ejecución de tareas en GPU. Esta heurística se puede incluir también como un módulo en tiempo de ejecución ya que no modifica los *kernels* originales y tiene una penalización insignificante en el tiempo de ejecución.

Se han llevado a cabo varios experimentos que muestran la confianza y robustez de este nuevo enfoque. Para este propósito, se han evaluado tres arquitecturas GPU diferentes (Kepler, Maxwell y Pascal), usando varios *kernels* reales procedentes de los SDK de CUDA y Rodinia. Se ha llevado a cabo un análisis estadístico para valorar la confianza estadística de la heurística NEH-GPU y mostrar su ventaja sobre otras heurísticas. Además, para valorar la escalabilidad de las heurísticas, se ha variado el número de tareas en los experimentos. En todos ellos, la heurística NEH-GPU ha obtenido los resultados más cercanos al mejor valor de *makespan* obtenido por cualquier heurística.

5.2. 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. Gómez-Luna, J.M. González-Linares, N. Guil, Asynchronous tasks queue scheme on gpu. Jornadas de Paralelismo. Sarteco, 2014.

  Implementación sobre NVIDIA-CUDA de un esquema de colas de tareas para entornos donde la CPU y el acelerador tienen distintos espacios de memoria. Este esquema es capaz de ejecutar concurrentemente transferencias de memoria con computación, adaptando el flujo de trabajo entre ambos dispositivos, y controlando el nivel de particionamiento de las cargas de trabajo. Esta solución ha sido comparada con una solución basada en CUDA streams, demostrando que la solución propuesta es capaz de obtener un mejor balanceo de carga.

- W Hwu Wen-mei.Heterogeneous System Architecture: A new compute platform infrastructure, chapter Aplication Uses Cases: Platform Atomics. Ed. Morgan Kaufmann, 2015, ISBN 978-0-12-800386-2

  Desarrollo de un esquema productor/consumidor en sistemas donde la CPU y el acelerador pueden compartir el mismo espacio de direcciones. En este trabajo se pone de manifiesto la consistencia de memoria proporcionada por las *Platforms Atomics* en arquitecturas heterogéneas. Para este fin se evalúan las *Platforms Atomics* en tres tipos de aplicaciones donde esta característica puede ser beneficiosa. Los resultados obtenidos por el esquema propuesto en este trabajo demuestran que las implementaciones que hacen

uso de *Platforms Atomics* consiguen un mejor balanceo de carga entre la CPU y el acelerador.

- A.J. Lázaro-Muñoz, J.M. González-Linares, J. Gómez-Luna, N. Guil, Efficient OpenCL-based concurrent tasks offloading on accelerators, Procedia Computer Science, Volume 108, 2017, Pages 2353-2357, ISSN 1877-0509, `https://doi.org/10.1016/j.procs.2017.05.113`.

  Desarrollo en entornos OpenCL de una solución software para optimizar la concurrencia entre transferencias de memoria y computación. Este esquema ha sido desarrollado para entornos multihilo donde cada hilo CPU puede enviar tareas al acelerador. El sistema propuesto implementa una heurística que hace uso de un modelo temporal de ejecución para tareas concurrentes. Esta es capaz de predecir un orden de planificación de tareas cercano al óptimo que reduce significativamente el tiempo total de ejecución, incluyendo las transferencias de datos. Esta solución ha sido evaluada en arquitecturas como NVIDIA, AMD e Intel Xeon Phi, obteniendo un alto grado de solapamiento entre computación y transferencias de memoria.

- A. J. Lázaro-Muñoz, J.M. González-Linares, J. Gómez-Luna, N. Guil, Improving tasks throughput on accelerators using opencl command concurrency, Version 2, Date 2018-07-01, `http://arxiv.org/abs/1806.10113v2`

  Este artículo es una extensión de *Efficient OpenCL-based concurrent tasks offloading on accelerators* publicado en Procedia Computer Science (ICCS 2017), Vol. 108, pp. 1353-1357, 2017, e incluye material que hubo que eliminar por restricciones de espacio.

- A.J. Lázaro-Muñoz, J.M. González-Linares, J. Gómez-Luna, N. Guil, A tasks reordering model to reduce transfers overhead on GPUs, Journal of Parallel and Distributed Computing, Available online 29 June 2017, ISSN 0743-7315, `https://doi.org/10.1016/j.jpdc.2017.06.015`.

  Desarrollo en entornos NVIDIA-CUDA de una solución software que se presenta como una alternativa al planificador hardware denominado Hyper-Q presente en los actuales aceleradores de NVIDIA. Hyper-Q habilita distintas colas hardware de tareas para la comunicación con el acelerador, las cuales pueden ser planificadas de forma independiente. El esquema presentado en este trabajo es capaz de optimizar la concurrencia entre transferencias de datos y computación en escenarios multihilo, donde cada hilo CPU puede enviar tareas al acelerador. Un modelo temporal de ejecución es utilizado

por esta solución para conseguir un orden de planificación de tareas cercano al óptimo y al mismo tiempo capaz de mejorar al utilizado por el planificador hardware HyperQ en los actuales aceleradores de NVIDIA.

- 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, relación de artículos aceptados: `https://vecpar2018.ncc.unesp.br/index.php/approved-papers`

  Aplicación de la teoría de planificación en la ejecución concurrente de tareas en 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.

5.3. Líneas futuras de investigación

Durante esta tesis se ha estudiado la planificación de grupos de tareas sobre aceleradores y han surgido algunos aspectos que podrían constituir nuevas líneas de investigación.

5.3.1. Restricciones de memoria en el acelerador

En todo el trabajo desarrollado se ha considerado que la memoria en el acelerador es suficientemente grande como para almacenar todos los datos que necesitan todas las tareas que se quieren planificar. Esta suposición puede ser cierta en muchas ocasiones pero impone una restricción que no se debe ignorar ya que para considerar cargas de trabajo más grandes sería necesario incluir algún mecanismo para liberar memoria durante la ejecución del grupo de tareas. Desgraciadamente, en la actualidad la gestión de la memoria en el acelerador por parte del driver implica que, cuando llega un comando de liberación de la memoria, es necesario sincronizar todos los comandos enviados con anterioridad al dispositivo. Esta

sincronización impide que los comandos planificados posteriormente se pueden ejecutar concurrentemente con los que ya estaban.

Para abordar este problema se pueden plantear diversas soluciones. Por ejemplo, se podría implementar un nuevo gestor de memoria que haga innecesario sincronizar cuando llegue una liberación. Este gestor de memoria tendría que encargarse de asignarle memoria a las tareas y solucionar la fragmentación que surgiría conforme las tareas liberan la memoria, bien mediante alguna técnica de compactación o bien con una sincronización cuando la fragmentación impida iniciar nuevas tareas.

Otra solución sería adoptar el mecanismo de memoria unificada proporcionado a partir de CUDA 6.0. Este mecanismo hace innecesarias las transferencias explícitas de datos entre el dispositivo y el procesador, pero a cambio suponen una penalización importante en el rendimiento. Esta solución requeriría cambiar el modelo de ejecución de tareas, ya que no habría comandos *HtD* y *DtH*, y buscar como mejorar la ejecución concurrente de los kernels.

5.3.2. Aplicación de la Teoría de Planificación

En esta tesis hemos modelado la planificación de tareas en un acelerador como un problema *Flow Shop* pero la Teoría de la Planificación proporciona otras formas de organizar las máquinas y los trabajos que podrían adaptarse bien a otros escenarios, como por ejemplo granjas de GPUs o sistemas heterogéneos más complejos. Este enfoque permitiría plantear nuevos algoritmos de planificación en tiempo real que se podrían incorporar a sistemas ya existentes como MPS o rCUDA. Otra posibilidad sería utilizar este enfoque para estudiar aplicaciones ya existentes y buscar el reparto de tareas más adecuado para su ejecución eficiente en un sistema heterogéneo. Una tercera opción consistiría en cambiar la función objetivo y, en lugar de minimizar el *makespan*, plantear otras posibilidades como un reparto equitativo de los recursos (*fairness*) o garantizar unas prestaciones de calidad de servicio (*QoS*).

5.3.3. Ejecución concurrente de kernels

El sistema que hemos desarrollado deshabilita la posibilidad de ejecutar concurrentemente los kernels (CKE) ya que no se obtienen buenos resultados en cuanto al tiempo de ejecución y dificulta enormemente el análisis. No obstante, puede haber situaciones como por ejemplo cuando se requiere usar una política de planificación basada en la equidad, en las cuales el uso de CKE estaría justi-

ficado. Para ello sería necesario desarrollar un sistema de lanzamiento de kernels que permita controlar los recursos hardware que consumirá el kernel, y un modelo capaz de predecir el comportamiento de ese kernel cuando se está ejecutando concurrentemente con otro (u otros).

Bibliografía

[1] Top500 supercomputer list. https://www.top500.org/, 2018. (Cited on page 1)

[2] 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* (New York, NY, USA, 1995), SPAA '95, ACM, pp. 95–105. (Cited on pages 51 and 61)

[3] Allahverdi, A., Ng, C., Cheng, T., and Kovalyov, M. Y. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research 187*, 3 (jun 2008), 985–1032. (Cited on page 121)

[4] AMD. Amd graphics cores next (gcn) architecture. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide.pdf, December 2014. (Cited on pages 52, 56 and 59)

[5] AMD. Compute cores. https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf, January 2014. (Cited on page 5)

[6] AMD. White paper — amd graphics cores next (gcn) architecture. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf, June 2012. (Cited on page 16)

[7] Arora, N. S., Blumofe, R. D., and Plaxton, C. G. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems 34*, 2 (2001), 115–144. (Cited on page 82)

[8] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. Starpu: A unified platform for task scheduling on heterogeneous multicore

architectures. *Concurr. Comput. : Pract. Exper. 23*, 2 (Feb. 2011), 187–198. (Cited on pages 65 and 86)

[9] Awatramani, M., Zambreno, J., and Rover, D. Increasing gpu throughput using kernel interleaved thread block scheduling. In *2013 IEEE 31st International Conference on Computer Design (ICCD)* (Oct 2013), pp. 503–506. (Cited on page 53)

[10] Blumofe, R., and Leiserson, C. Scheduling multithreaded computations by work stealing. *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), 1–29. (Cited on page 82)

[11] 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* (May 2013), IEEE, pp. 1097–1106. (Cited on page 50)

[12] Cánovas, D. G., García, L.-P., and Cuenca, J. On optimization techniques for the matrix multiplication on hybrid cpu+ gpu platforms. *Annals of Multicore and GPU Programming 2*, 1 (2014), 1–8. (Cited on page 51)

[13] Cederman, D., Tsigas, P., Balancing, D. L., and Work-stealing, U. Dynamic Load Balancing Using Work-Stealing. In *GPU Computing Gems*, vol. 1. Elsevier Inc., 2012, pp. 485–500. (Cited on page 82)

[14] Chatterjee, S., Grossman, M., Sbîrlea, A., and Sarkar, V. Dynamic task parallelism with a GPU work-stealing runtime system. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2013), vol. 7146 LNCS, pp. 203–217. (Cited on page 82)

[15] 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 *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on* (Oct 2009), pp. 44–54. (Cited on page 95)

[16] Chen, L., Villa, O., Krishnamoorthy, S., and Gao, G. R. Dynamic Load Balancing on Single- and Multi-GPU Systems. (Cited on page 83)

[17] Chrysos, G. Intel xeon phi coprocessor (codename knights corner). In *2012 IEEE Hot Chips 24 Symposium (HCS)* (Aug 2012), pp. 1–31. (Cited on page 19)

[18] 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. *Commun. ACM 39*, 11 (Nov. 1996), 78–85. (Cited on pages 51 and 61)

[19] Duato, J., Pena, A. J., Silla, F., Mayo, R., and Quintana-Orti, E. S. Performance of cuda virtualized remote gpus in high performance clusters. In *2011 International Conference on Parallel Processing* (Sept 2011), pp. 365–374. (Cited on page 2)

[20] Duato, J., Peña, A. J., Silla, F., Mayo, R., and Quintana-Ortí, E. S. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing Simulation* (June 2010), pp. 224–231. (Cited on page 86)

[21] Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters 21*, 02 (2011), 173–193. (Cited on pages 65 and 86)

[22] Framinan, J., Gupta, J., and Leisten, R. A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. *Journal of the Operational Research Society 55* (2004), 1243–1255. (Cited on page 121)

[23] Garey, M. R., Johnson, D. S., and Sethi, R. The Complexity of Flowshop and Jobshop Scheduling. *Mathematics of Operations Research 1*, 2 (1976), 117–129. (Cited on page 122)

[24] Gautier, T., Lima, J. V., Maillard, N., and Raffin, B. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on* (2013), IEEE, pp. 1299–1308. (Cited on page 86)

[25] Ghiasi, S., Keller, T., and Rawson, F. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd Conference on Computing Frontiers* (2005), ACM, pp. 199–210. (Cited on page 84)

[26] Gómez-Luna, J., González-Linares, J. M., Benavides, J. I., and Guil, N. Performance models for asynchronous data transfers on consumer Graphics Processing Units. *J. Parallel Distrib. Comput. 72*, 9 (sep 2012), 1117–1126. (Cited on page 52)

[27] Gómez-Luna, J., Sung, I.-J., Lázaro-Muñoz, A., Chung, W.-H., González-Linares, J., and Guil, N. *Heterogeneous System Architecture: A New Compute Platform Infrastructure*. Morgan Kaufmann, 2015, ch. Application Use Cases: Platform Atomics. (Cited on page 83)

[28] Graham, R., Lawler, E., Lenstra, J., and Kan, A. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. *Annals of Discrete Mathematics 5*, C (1979), 287–326. (Cited on page 117)

[29] 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`, April 2017. (Cited on page 5)

[30] Intel. Intel xeon phi coprocessor developer quick start guide. `http://www.serc.iisc.in/facilities/wp-content/uploads/2015/02/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf`, May 2016. (Cited on pages 18 and 52)

[31] Jiménez, V. J., Vilanova, L., Gelado, I., Gil, M., Fursin, G., and Navarro, N. Predictive runtime code scheduling for heterogeneous architectures. *HiPEAC 9* (2009), 19–33. (Cited on page 84)

[32] Johnson, S. Optimal Two- and Three-Stage Production Schedules With Set-up Time Included. *Naval Research Logistics Quarterly 1* (1954), 61–68. (Cited on page 122)

[33] 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)* (Oct 2013), pp. 15–22. (Cited on page 52)

[34] Khronos Group. Opencl 2.0 api specification. `https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`, October 2014. (Cited on pages 2, 31, 52 and 60)

[35] 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* (March 2015), pp. 651–658. (Cited on page 52)

[36] Kumar, R., Tullsen, D. M., Jouppi, N. P., and Ranganathan, P. Heterogeneous chip multiprocessors. *Computer 38*, 11 (Nov. 2005), 32–38. (Cited on page 83)

[37] Lázaro-Muñoz, A., González-Linares, J., Gómez-Luna, J., and Guil, N. Efficient opencl-based concurrent tasks offloading on accelerators. In *Proceedings of the International Conference on Computer Science* (2017), ICCS '17, pp. 2353–2357. (Cited on page 86)

[38] Lázaro-Muñoz, A., González-Linares, J., López-Albelda, B., and Guil, N. A scheduling theory framework for gpu tasks efficient execution. In *13th International Meeting on High Performance Computing for Computational Science* (2018), VECPAR18. (Cited on page 117)

[39] Lázaro-Muñoz, A. J., Gómez-Luna, J., González-Linares, J., and Guil, N. Asynchronous tasks queue scheme on gpu. In *Jornadas de Paralelismo* (2014), Sarteco. (Cited on page 83)

[40] Lázaro-Muñoz, A. J., González-Linares, J., Gómez-Luna, J., and Guil, N. A tasks reordering model to reduce transfers overhead on GPUs. *Journal of Parallel and Distributed Computing 109* (nov 2017), 258–271. (Cited on pages 106, 117, 120, 124 and 127)

[41] Lázaro-Muńoz, A. J., González-Linares, J. M., Gómez-Luna, J., and Guil, N. Improving tasks throughput on accelerators using opencl command concurrency. `http://arxiv.org/abs/1806.10113v2`, 2018. (Cited on page 86)

[42] Lemeire, J., Cornelis, J. G., and Segers, L. Microbenchmarks for gpu characteristics: The occupancy roofline and the pipeline model. In *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on* (2016), IEEE, pp. 456–463. (Cited on page 52)

[43] 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. *CoRR abs/1603.08619* (2016). (Cited on page 53)

[44] 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 (March 2015), 748–760. (Cited on page 82)

[45] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro 28*, 2 (3 2008), 39–55. (Cited on page 7)

[46] Liu, B., Qiu, W., Jiang, L., and Gong, Z. Software pipelining for graphic processing unit acceleration: Partition, scheduling and granularity.

International Journal of High Performance Computing Applications (2015). (Cited on pages 52 and 65)

[47] Luk, C.-K., Hong, S., and Kim, H. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), ACM, pp. 45–55. (Cited on page 84)

[48] Margiolas, C., and O'Boyle, M. F. P. Portable and transparent host-device communication optimization for gpgpu environments. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (New York, NY, USA, 2014), CGO '14, ACM, pp. 55:55–55:65. (Cited on page 60)

[49] Meng, J., Morozov, V. a., Kumaran, K., Vishwanath, V., and Uram, T. D. GROPHECY: GPU performance projection from CPU code skeletons. *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2011), 1–11. (Cited on pages 52 and 65)

[50] Navarro, A., Vilches, A., Corbera, F., and Asenjo, R. Strategies for maximizing utilization on multi-cpu and multi-gpu heterogeneous architectures. *The Journal of Supercomputing 70*, 2 (2014), 756–771. (Cited on page 82)

[51] Nawaz, M., Enscore, 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 pages 123 and 124)

[52] NVIDIA. Nvidia next generation compute architecture: Fermi. `https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, April 2010. (Cited on page 8)

[53] NVIDIA. Kepler gk100 architecture whitepaper. `https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, April 2012. (Cited on page 10)

[54] NVIDIA. Nvidia geforce gtx 1080. `https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf`, April 2016. (Cited on page 14)

[55] NVIDIA. Nvidia tegra k1. a new era in mobile computing. https://www.nvidia.com/content/PDF/tegra_white_papers/tegra-K1-whitepaper.pdf, January 2014. (Cited on page 5)

[56] NVIDIA. Nvidia geforce gtx 980. https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF, January 2015. (Cited on page 12)

[57] NVIDIA. Cuda multi-process service. "https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, March 2015. (Cited on pages 2 and 121)

[58] NVIDIA. Nvidia tesla v100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/Volta-Architecture-Whitepaper-v1.0.pdf, May 2017. (Cited on page 15)

[59] NVIDIA. Cuda programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, September 2015. (Cited on pages 2, 7, 20, 51, 52, 54, 55, 60 and 86)

[60] NVIDIA. Cuda samples. http://docs.nvidia.com/cuda/cuda-samples/index.html, September 2015. (Cited on page 4)

[61] Palmer, D. S. 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 (mar 1965), 101–107. (Cited on page 123)

[62] 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), ACM, pp. 527–540. (Cited on page 82)

[63] Pinedo, M. *Scheduling. Theory, algorithms, and systems. With CD-ROM. 3rd ed.* 01 2008. (Cited on page 119)

[64] Planas, J. *Programming Models and Scheduling Techniques for Heterogeneous Architectures.* PhD thesis, 2015. (Cited on page 84)

[65] Ravi, V. T., Becchi, M., Agrawal, G., and Chakradhar, S. Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th international symposium on*

High performance distributed computing (2011), ACM, pp. 217–228. (Cited on page 83)

[66] Ruiz, R., and Maroto, C. A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research 165*, 2 (sep 2005), 479–494. (Cited on page 122)

[67] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 18:1–18:15. (Cited on page 18)

[68] Shee, S. L., and Parameswaran, S. Design methodology for pipelined heterogeneous multiprocessor system. In *Proceedings of the 44th annual Design Automation Conference* (2007), ACM, pp. 811–816. (Cited on page 84)

[69] Sim, J., Dasgupta, A., Kim, H., and Vuduc, R. A performance analysis framework for identifying potential benefits in GPGPU applications. *ACM SIGPLAN Notices 47*, 8 (2012), 11. (Cited on pages 52 and 65)

[70] Sodani, A., Gramunt, R., Corbal, J., Kim, H. S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y. C. Knights landing: Second-generation intel xeon phi product. *IEEE Micro 36*, 2 (Mar 2016), 34–46. (Cited on page 19)

[71] Souza, P., Newburn, C., and Borges, L. Heterogeneous architecture library. In *Second EAGE Workshop on High Performance Computing for Upstream* (2015). (Cited on page 53)

[72] Stanisic, L., Thibault, 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 51)

[73] Stroustrup, B. Programming: Principles and practice using c++, November 2008. (Cited on page 86)

[74] Suzuki, T., Nukada, A., and Matsuoka, S. *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, ch. Efficient Execution of Multiple CUDA Applications Using Transparent Suspend, Resume and Migration, pp. 687–699. (Cited on page 53)

[75] Taillard, E. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research 47* (1990), 65–74. (Cited on pages 123 and 124)

[76] Takizawa, H., Hirasawa, S., Sugawara, M., Gelado, I., Kobayashi, H., and Hwu, W.-m. W. Optimized data transfers based on the opencl event management mechanism. *Sci. Program. 2015* (Jan. 2016), 2:2–2:2. (Cited on page 83)

[77] Totoni, E., Dikmen, M., and Garzarán, M. J. Easy, fast, and energy-efficient object detection on heterogeneous on-chip architectures. *ACM Transactions on Architecture and Code Optimization (TACO) 10*, 4 (2013), 45. (Cited on page 83)

[78] Tukey, J. W. *Exploratory Data Analysis*. 1977. (Cited on page 131)

[79] Tzeng, S., Lloyd, B., and Owens, J. D. A GPU task-parallel model with dependency resolution. *Computer 45*, 8 (2012), 34–41. (Cited on page 83)

[80] Valero-Lara, P., and Pelayo, F. L. Full-overlapped concurrent kernels. In *Architecture of Computing Systems. Proceedings, ARCS 2015-The 28th International Conference on* (2015), VDE, pp. 1–8. (Cited on page 83)

[81] van Werkhoven, B., Maassen, J., Seinstra, F., and Bal, H. Performance Models for CPU-GPU Data Transfers. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (May 2014), IEEE, pp. 11–20. (Cited on pages 51, 54 and 61)

[82] Vilches, A., Navarro, A., Asenjo, R., Corbera, F., Gran, R., and Garzarán, M. J. Mapping streaming applications on commodity multi-cpu and gpu on-chip processors. *IEEE Transactions on Parallel and Distributed Systems 27*, 4 (April 2016), 1099–1115. (Cited on page 83)

[83] Voran, T., Garcia, J., and Tufo, H. Evaluating intel's many integrated core architecture for climate science. In *Proceedings of TACC-Intel highly parallel computing symposium* (2012). (Cited on page 18)

[84] Wagner, H. M. An integer linear-programming model for machine scheduling. *Naval Research Logistics Quarterly 6*, 2 (jun 1959), 131–140. (Cited on page 122)

[85] Wang, J., Rubin, N., Sidelnik, A., and Yalamanchili, S. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus. *ACM SIGARCH Computer Architecture News 43*, 3 (2016), 528–540. (Cited on page 83)

[86] Wang, L., Huang, M., and El-Ghazawi, T. Exploiting concurrent kernel execution on graphic processing units. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on* (July 2011), pp. 24–32. (Cited on page 53)

[87] Wende, F. *Dynamic Load Balancing on Massively Parallel Computer Architectures*. PhD thesis, 2013. (Cited on page 83)

[88] Wende, F., Cordes, F., and Steinke, T. On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. *Symposium on Application Accelerators in High-Performance Computing* (2012), 74–83. (Cited on pages 53 and 83)

[89] Wende, F., Steinke, T., and Cordes, F. Multi-threaded kernel offloading to gpgpu using hyper-q on kepler architecture. (Cited on page 83)

[90] 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)* (June 2016), pp. 230–242. (Cited on page 82)

[91] Zhong, J., and He, B. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst. 25*, 6 (June 2014), 1522–1532. (Cited on page 53)

[92] 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 Trans. Archit. Code Optim. 13*, 4 (Dec. 2016), 35:1–35:25. (Cited on page 51)


UNIVERSIDAD
DE MÁLAGA