Departament d'Informàtica de Sistemes i Computadors
Universitat Politècnica de València

Mejora de las prestaciones de la GPU mediante planificación de aplicaciones concurrentes

Trabajo Fin de Master

Máster en Ingeniería de Computadores

Autor: David Baselga Masià

Directores: Salvador Petit y
Julio Sahuquillo

6 de septiembre de 2018

Resumen

En la actualidad, las *Graphics Processing Units* (GPU) no sólo se dedican a la aceleración de gráficos, sino que su uso se ha extendido a la ejecución de aplicaciones de propósito general, surgiendo así el paradigma conocido como *General Purpose computing on Graphics Processing Units* (GPGPU). Este paradigma se concibió en un contexto donde una única aplicación disponía de todos los recursos hardware de la GPU, al contrario que las *Central Processing Units* (CPU) donde se utilizan planificadores para decidir qué aplicaciones se ejecutan concurrentemente y, por tanto, comparten los recursos del sistema. Por este motivo, las GPU actuales no ofrecen un soporte adecuado a la ejecución concurrente de aplicaciones, lo cual y tal y como se demuestra en este trabajo, puede tener un impacto significativo en la utilización de sus recursos computacionales y, por ende, en las prestaciones finales del sistema.

En este trabajo se caracterizan una serie de aplicaciones científicas GPGPU, mostrando diferencias que pueden superar un orden de magnitud en el uso de recursos como el ancho de banda de memoria o las unidades de cómputo. Para estudiar la sensibilidad de las aplicaciones a la contención en el acceso los recursos compartidos, se han desarrollado *microbenchmarks* que interfieren en el uso de estos recursos. Los resultados demuestran que estas interferencias pueden llegar en algunos casos a incrementar el tiempo de ejecución de la aplicación bajo estudio hasta un 80 %.

Con el objetivo de mejorar el soporte de las GPU a la ejecución concurrente así como las prestaciones de las aplicaciones que comparten la GPU, se propone una estructura para la planificación de aplicaciones. Se trata de una estructura que controla el uso de la GPU por parte de las aplicaciones para facilitar la implementación de algoritmos de planificación efectivos. Sobre esta estructura, y en base a los resultados de la caracterización, se ha desarrollado el algoritmo de planificación SHARK propuesto en este trabajo, el cual trata de minimizar las interferencias en la jerarquía de memoria.

Los experimentos realizados muestran que SHARK mejora las prestaciones de la ejecución concurrente en comparación un algoritmo de planificación agnóstico a las interferencias, siendo sus ventajas más evidentes cuanto mayor es el nivel de paralelismo. En algunos casos se alcanzan aceleraciones superiores al 40 %.

Palabras clave: Aplicaciones GPGPU, Cargas multiprograma, Planificación, NVIDIA

Resum

En l'actualitat, les *Graphics Processing Units* (GPU) no només es dediquen a l'acceleració de gràfics, sinó que el seu ús s'ha estès a l'execució d'aplicacions de propòsit general, sorgint així el paradigma conegut com *General Purpose computing on Graphics Processing Units* (GPGPU). Aquest paradigma es va concebre en un context on una única aplicació disposava de tots els recursos del hardware de la GPU, al contrari que les *Central Processing Units* (CPU) on s'empren planificadors per a decidir quines aplicacions s'executen concurrentment i, per tant, comparteixen els recursos del sistema. En altres paraules, les GPU actuals no donen un suport adequat a l'execució concurrent d'aplicacions, la qual cosa té un impacte en la utilització dels seus recursos computacionals i, per tant, en les prestacions globals del sistema.

En aquest treball es caracteritzen una sèrie d'aplicacions científiques GPGPU, mostrant diferències que poden superar un ordre de magnitud en l'ús de recursos com l'ample de banda de memòria o les unitats de còmput. Per a observar la sensibilitat de les aplicacions en l'accés als recursos compartits, s'han desenvolupat *microbenchmarks* que interfereixen en l'ús d'aquests recursos, arribant en alguns casos fins a un 80% d'increment en el temps d'execució.

Amb l'objectiu de millorar el suport de les GPU a l'execució concurrent així com les prestacions de les aplicacions que comparteixen la GPU, es proposa una estructura per desenvolupar planificadors d'aplicacions en GPU. Es tracta d'una estructura que controla l'ús de la GPU per part de les aplicacions per a poder així implementar algorismes de planificació. Sobre aquesta estructura, i sobre la base dels resultats de la caracterització, es desenvolupa l'algorisme de planificació SHARK proposat en aquest treball, el qual tracta de minimitzar les interferències en la jerarquia de memòria de la GPU.

Els experiments realitzats amb aquest algorisme mostren que SHARK millora les prestacions de l'execució concurrent en comparació un algorisme de planificació agnòstic a les interferències, sent els seus avantatges més evidents com més gran és el nivell de paral·lelisme. En alguns casos s'aconsegueixen acceleracions superiors al 40%.

Paraules clau: Aplicacions GPGPU, Càrregues multiprograma, Planificació, NVIDIA

Abstract

Nowadays, Graphics Processing Units (GPUs) are not only used for graphics acceleration but they are also being employed in general purpose computing. As a result, the General Purpose computing on Graphics Processing Units (GPGPU) paradigm emerged. This paradigm was initially developed in a context where a single application had available all the GPU's hardware resources. In constrast, Central Processing Units (CPUs) are managed by schedulers that select which applications run concurrently and thus share the system resources. In other words, current GPUs do not provide a mature support for concurrent GPGPU applications, which impacts on resource utilization and on performance.

In this work, several GPGPU applications are characterized, presenting differences among them that can be higher than an order of magnitude in the utilization of resources like memory bandwidth or computational units. To study application's sensitivity to shared resource contention, we develop microbenchmarks that interfere when accessing these resources. The results of the characterization study show that the execution time can grow by a 80% in some cases.

With the aim of improving concurrent execution support in GPUs and increasing system performance, we propose a novel scheduling framework for GPUs. This framework controls the GPU usage of applications and enables the implementation of new scheduling algorithms. In this scheduling framework, we develop SHARK, a scheduling algorithm that takes into account the results of the characterization study in order to minimize interferences in the memory hierarchy and improve performance.

Experimental results show that SHARK improves concurrent execution performance when compared to an interference-agnostic algorithm and that its benefits become more evident as execution parallelism increases. In some cases, the speedup is higher than 40%.

Keywords: GPGPU applications, Multiprogram workloads, Scheduling, NVIDIA

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo se introducen los conceptos básicos sobre los que se desarrollará este trabajo. Se comienza por una breve descripción y contextualización sobre el cómputo de propósito general en GPU (GPGPU). Acto seguido se procede a una explicación sobre las arquitecturas NVIDIA, centrándose en el caso de la Arquitectura Maxwell utilizada en el presente trabajo. En tercer lugar, se describe brevemente CUDA, el entorno software de NVIDIA que permite al programador desarrollar aplicaciones para GPU. Finalmente se detallan los problemas que se pretenden resolver y las contribuciones del trabajo.

1.1 *General Purpose computing on Graphics Processing Units* (GPGPU)

Con la finalidad de acelerar el procesamiento de imágenes y vídeo, se ideó un hardware especializado, las *Graphics Processing Unit* (GPU) o unidades de procesamiento gráfico. Las GPU se componen de un número masivo de unidades de cómputo. Aunque la potencia de cómputo de cada una de estas unidades es muy baja, especialmente si se compara con los procesadores convencionales o CPU, el hecho de que exista un número masivo, así como su eficiencia energética, llevó a la comunidad académica y la industria a interesarse por ellas. En la actualidad, las lineas comerciales de GPU ofrecen tarjetas optimizadas según las necesidades de su usuario, ya sean estos usuarios consumidores de entretenimiento, profesionales o científicos [1].

El motivo por el cual las GPU pueden ser aprovechadas para el uso científico es que el calculo de *frames* requiere de gran cantidad de cálculo vectorial, cuyo ámbito de aplicación trasciende este uso, siendo utilizado para resolver gran cantidad de problemas. En comparación, en una Unidad Central de Procesamiento (*Central Processing Unit* o CPU) la computación sobre matrices debe realizarse iterativamente haciendo uso de bucles anidados, lo que reduce las prestaciones [2].

De este modo, los dispositivos que soportan el paradigma GPGPU han mostrado ser útiles para la aceleración de aplicaciones con un alto grado de paralelismo [2]. Tal es su importancia en la computación de altas prestaciones (*HPC*) la mitad de las supercomputadoras del Top10 de la lista Top500 de Junio del 2018 [3], hacen uso de este hardware.

1.2 Arquitectura de las GPU NVIDIA

Las GPU actuales implementan varios multiprocesadores, denominados *Streaming Multiprocessor (SM)* en la arquitectura NVIDIA, con la capacidad de ejecutar múltiples instrucciones sobre varios múltiples datos simultáneamente. Esto se logra mediante la replicación de hardware. Así, cada multiprocesador contiene una o varias unidades de *Single Instruction Multiple Data (SIMD)*, a su vez compuestas, entre otros componentes, por unidades funcionales especializadas en varios tipos de operaciones, que se replican con el fin de reducir el número de ciclos necesarios para completar una misma operación en una multitud de datos escalares.

Cada multiprocesador incorpora unos recursos que se comparten entre los distintos *kernels* en ejecución. Estos recursos son los bancos de registros, las memorias cache de primer (L1) y segundo nivel (L2), un banco de memoria

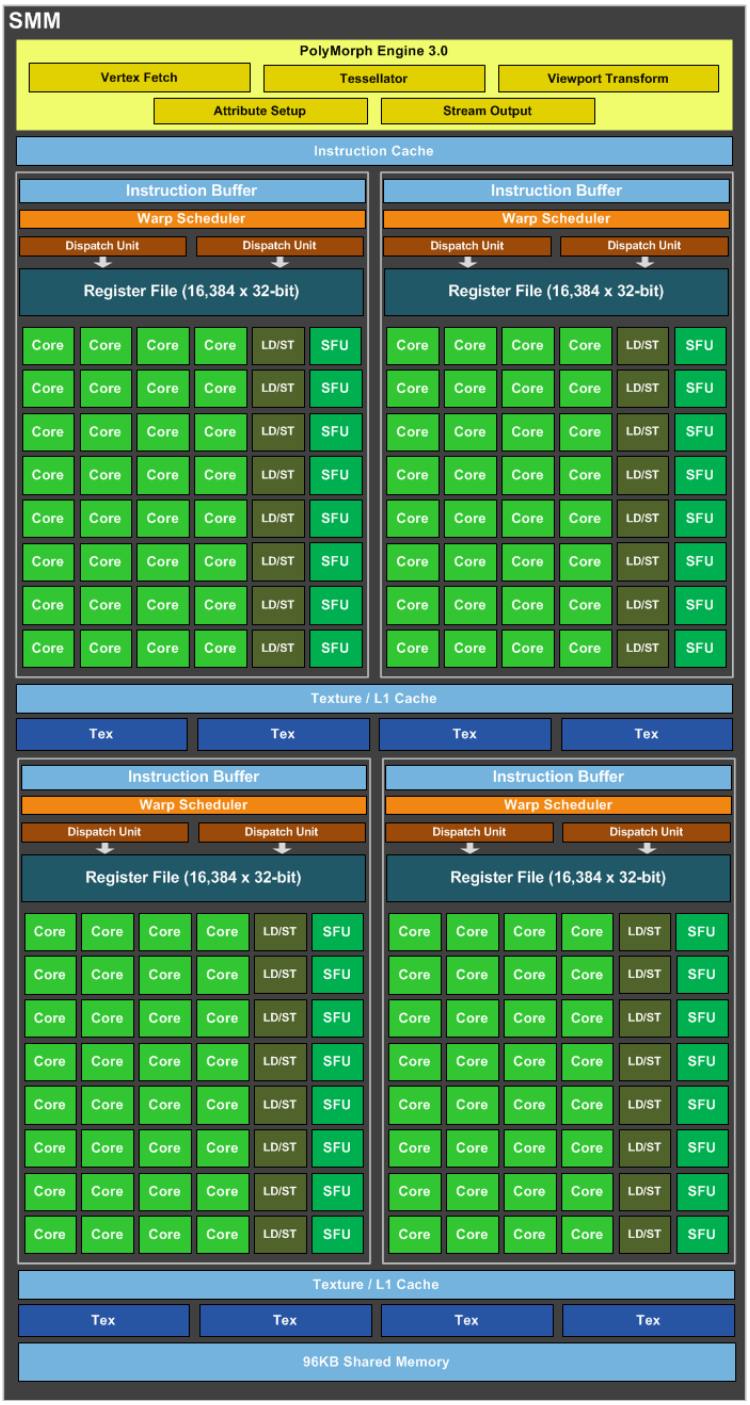


Figura 1.1: SM de arquitectura Maxwell de segunda generación. Fuente: NVIDIA Corporation.

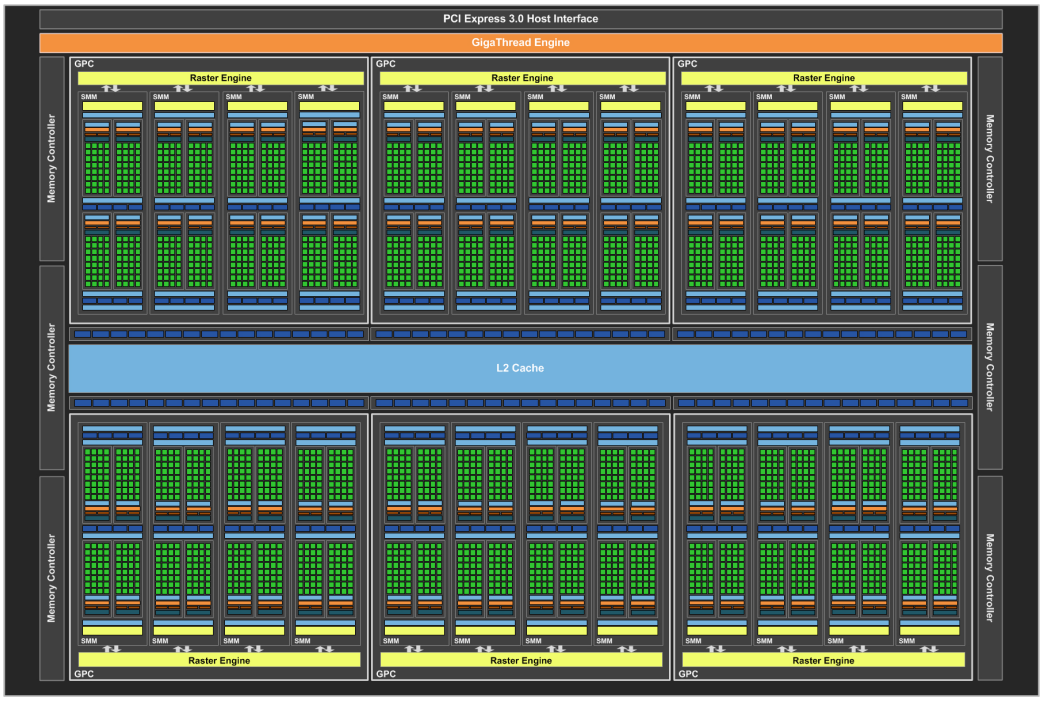


Figura 1.2: Tarjeta NVIDIA GeForce GTX Titan X. Fuente: NVIDIA Corporation.

local conocido como *shared memory* y las distintas unidades funcionales. En la figura 1.1 puede apreciarse un diagrama mostrando como se estructura un SM de arquitectura Maxwell de segunda generación.

Una GPU de NVIDIA implementa decenas de SM que comparten la memoria cache de segundo nivel (L2), que actúa como la de último nivel (LLC). Además, los SM comparten la memoria principal GDDR5 de la GPU que se accede a través de múltiples controladores de acceso a memoria. Todo esto se puede apreciar en la figura 1.2, la cual muestra los SM disponibles en una tarjeta NVIDIA GeForce GTX Titan X y los recursos de la jerarquía de memoria que estos comparten.

1.3 CUDA

Compute Unified Device Architecture o simplemente CUDA es una plataforma de programación paralela y su correspondiente API desarrollados por NVIDIA para programar sus dispositivos. CUDA se centra en la programación de algoritmos altamente paralelizables y en el uso eficiente de las capacidades de una GPU por parte de estos algoritmos. Esta plataforma permite

programar en diversos lenguajes, desde C hasta Java, pasando por otros como Fortran o Python. Mediante la plataforma CUDA, el programador puede: i) reservar espacio en la memoria principal de la GPU, ii) realizar transferencias de datos y código entre la memoria principal del computador y la de la GPU, y iii) programar el código o kernel que se ejecutará en la GPU.

Un kernel está formado por una multitud de *threads* o hilos. Estos hilos se agrupan en mallas (*grids*) que a su vez se subdividen en bloques (*blocks*), cada uno de los cuales se asigna a un SM. El número de malla y bloque, así como la posición concreta de un thread dentro de su bloque es el factor que determina la posición de los datos en registros y memoria accedidos por cada thread durante la ejecución [4]. Finalmente, dentro del SM, los threads de un bloque se reparten en *warps*, donde cada thread de un warp se ejecuta en una unidad funcional diferente [5].

1.4 Problema a resolver

Originalmente, las GPU fueron diseñadas para acelerar la ejecución de los kernels de una misma aplicación. Por otro lado, las aplicaciones, que normalmente necesitan ejecutar múltiples kernels, los lanzaban a la GPU de manera secuencial.

Sin embargo, a medida que las GPU integraban más SM con un mayor número de núcleos quedó patente que un lanzamiento de los kernels secuencialmente limitaba oportunidades para explotar el paralelismo en la ejecución de los kernels independientes entre si. Es por ello que NVIDIA incorporó en CUDA la posibilidad de ejecutar varios kernels en paralelo.

Por defecto, una GPU NVIDIA actual sólo puede ejecutar en paralelo varios kernels de un mismo *contexto*, donde un *contexto* (en terminología CUDA) corresponde a una aplicación [6]. Con el incremento aún mayor de las capacidades de cómputo de las GPU está limitación también se ha demostrado demasiado restrictiva, ya que muchas aplicaciones son incapaces por sí solas de aprovechar todos los recursos de cómputo de la GPU, lo que lleva a su infrautilización.

Es decir, actualmente las GPU disponen de suficientes recursos para ejecutar más de una aplicación al mismo tiempo, por lo que la planificación de la ejecución de las aplicaciones en la GPU, es decir, elegir qué aplicaciones comparten los recursos de la GPU en cada momento pasa a ser un aspecto clave de diseño del sistema. La planificación puede tener diferentes objetivos, como maximizar las prestaciones, la utilización de la GPU, minimizar el consumo energético, la equidad, un mínimo de calidad de servicio, etc. En este trabajo, nuestro objetivo principal es minimizar el tiempo de ejecución

de los kernels.

Como en las CPU, elegir qué aplicaciones deben ejecutarse en una GPU para conseguir un cierto objetivo no es un problema sencillo. Esto se debe a que el lanzamiento de aplicaciones a la GPU sin un control adecuado puede generar interferencias y contención en el acceso a los recursos compartidos que limiten las prestaciones alcanzables. Así, es necesario que el planificador tenga en cuenta las características de las aplicaciones en lo que respecta a sus necesidades y consumo de recursos. Nótese que es posible que una aplicación consuma una gran cantidad de un recurso determinado pero que en realidad *no la necesite*, ya que sus prestaciones pueden ser poco sensibles a la abundancia o escasez de ese recurso. Este es el caso, por ejemplo, de aplicaciones con poca localidad temporal que generan muchos fallos de cache en la LLC, consumiendo una gran cantidad de líneas de cache y expulsando a otras aplicaciones de la LLC, pero sin hacer un uso efectivo de la cache ocupada. Por ello, antes de desarrollar un planificador para GPU que maximice las prestaciones, es necesario caracterizar el comportamiento de las aplicaciones que pueden compartir la GPU.

En resumen, debido a las ingentes capacidades de cómputo de una GPU actual, es muchas veces necesario que las aplicaciones la compartan para evitar que sea infrautilizada. Sin embargo, la interferencia de las aplicaciones en los recursos compartidos implica que para elegir qué aplicaciones deben ejecutarse en cada momento sea necesario caracterizar su comportamiento. Una vez caracterizadas las aplicaciones, la información obtenida podrá ser utilizada para el desarrollo de algoritmos de planificación que persigan un determinado objetivo, como la productividad global del sistema.

1.5 Contribuciones

Este trabajo realiza las siguientes contribuciones:

- En primer lugar, se desarrolla una estructura de planificación de aplicaciones para GPU que permite que varias aplicaciones compartan la GPU variando el nivel de paralelismo y el algoritmo de planificación utilizado. Esta estructura se ha utilizado en una GPU NVIDIA Titan X con arquitectura Maxwell de segunda generación.

- En segundo lugar, se realiza una caracterización de la ejecución en solitario de diversas aplicaciones científicas que demuestra que existen diferencias sustanciales entre ellas en lo que respecta a la utilización de recursos, lo que ofrece oportunidades para maximizar las prestacio-

nes mediante algoritmos de planificación que tengan en cuenta estas diferencias.

- En tercer lugar, mediante el uso de *microbenchmark* se realiza una caracterización de la ejecución concurrente de las aplicaciones. Esta caracterización muestra cómo las distintas interferencias en los recursos compartidos afectan a las prestaciones de cada aplicación, llegando a superar en algunos casos el 80 % de incremento en el tiempo de ejecución.

- Finalmente, se propone un algoritmo de planificación que tiene en cuenta el consumo de la LLC por las distintas aplicaciones y se evalúan sus prestaciones con respecto a un algoritmo de planificación por defecto.

Capítulo 2

Antecedentes y Trabajo Relacionado

Este capítulo se centra en describir el contexto en el que este trabajo se desarrolla, introduciendo las tecnologías de NVIDIA que permiten actualmente la ejecución concurrente de kernels, así como sus limitaciones. Además, se referencian e introducen trabajos con contenidos afines al presente trabajo y que han sido estudiados previamente al desarrollo del mismo.

2.1 Antecedentes

En esta sección se discuten las tecnologías Hyper-Q y MPS, las cuales se combinan en este trabajo para permitir ejecutándose en la GPU kernels de varias aplicaciones en paralelo.

2.1.1 NVIDIA Hyper-Q

Con la finalidad de poder solapar las transferencias entre la memoria de la CPU y la interna de la GPU, así como permitir que una aplicación pudiese lanzar dos o más kernels sin falsas dependencias, NVIDIA introdujo Hyper-Q [7] en su arquitectura Kepler. Esta tecnología implementa 32 colas de trabajo (*work queues*) independientes en lugar de la única con la que se contaba en arquitecturas NVIDIA previas.

Hyper-Q permite que los kernels de un contexto o proceso pueden compartir la GPU a través de la gestión de estas colas sin tener que esperar a que finalicen las transferencias relativas a los kernels lanzados previamente. En el momento en que se introdujo, esta tecnología supuso una mejora (*speedup*) de hasta un *2.5x* [8].

Sin embargo, Hyper-Q por sí sola no permite lanzar kernels de distintos procesos que compartan la GPU, lo que resulta un inconveniente para aquellas aplicaciones que se componen de varios procesos que pueden ejecutarse en el mismo sistema anfitrión de la GPU dependiendo de la configuración del sistema y del tamaño de la carga. Este es el caso, por ejemplo, de algunas aplicaciones paralelas GPGPU que se comunican utilizando el estándar MPI (*Message Passing Interface*).

2.1.2 MPS

A partir de la versión 3.5 de la arquitectura Kepler, CUDA incorpora el servicio MPS (*Multi Process Service*)[9], cuya principal finalidad es permitir que aplicaciones multiproceso MPI puedan ser aceleradas con la GPU. Este servicio es un software que captura los kernels que van a ser lanzados a la GPU por diferentes procesos y los asigna a un solo contexto de la GPU utilizando una técnica denominada *context funneling* [10].

Aunque MPS es un software ideado para aplicaciones MPI, también puede ser utilizado para que todo tipo de aplicaciones independientes entre sí compartan la GPU, lo que abre la posibilidad de planificar la ejecución de sus kernels. Sin embargo, el soporte a la ejecución concurrente de los kernels de diversas aplicaciones en la misma GPU es aún inmaduro ya que no incluye ni aislamiento de procesos ni protección de memoria dentro del mismo contexto,

lo que implica que el fallo de un kernel en un determinado momento puede terminar con toda la carga que se ejecuta en la GPU. A pesar de estos inconvenientes, que limitan la aplicación de esta técnica en entornos productivos, es lo suficientemente flexible para permitir la investigación académica sobre algoritmos de planificación.

2.2 Trabajo relacionado

Este trabajo realiza un estudio de caracterización de interferencias en los recursos compartidos y cuyos resultados se tienen en cuenta para la planificación de la ejecución concurrente de kernels en la GPU. Esta sección resume algunos de los trabajos más relacionados con estas temáticas.

2.2.1 Ejecución concurrente de kernels en la GPU

Como se expone en la anterior sección, actualmente, la ejecución concurrente de kernels de distintos procesos tiene como principal finalidad permitir que aplicaciones MPI puedan ser aceleradas conjuntamente. El diseño de esta funcionalidad, por lo tanto, no contempla el uso de aplicaciones con otras finalidades y esto tiene como consecuencia, entre otros aspectos, una excesiva simplificación de las decisiones de planificación. *Cruz. et al* [11] demuestran que las decisiones de planificación que toma MPS por defecto dependen notablemente del orden en el que los kernels son lanzados por las aplicaciones. Para explotar este hecho, desarrollan un planificador que reordena el lanzamiento de los kernels de modo que la ocupación de los recursos de la GPU se maximice. Los resultados obtenidos por este trabajo son muy significativos, logrando mejoras de hasta un 67 % en *Average Normalized Turnaround Time* (ANTT) y un 40 % del *System Throughtput* (STP) véase la definición de estas métricas de prestaciones en la Sección 4.3).

Otras consecuencia del diseño es el tratamiento de los datos de distintas aplicaciones. Mientras que una aplicación MPI tiende a utilizar los mismos datos y los mismos patrones de acceso en todos sus procesos, distintas aplicaciones pueden diferir enormemente en su comportamiento. *Ausavarungnirum. et al* [12] realiza un análisis basándose en estas observaciones y el resultado es que una parte importante de las causas del bajo rendimiento de la GPU en ejecución concurrente se debe a una pobre implementación de los mecanismos de memoria virtual de las mismas. El diseño, que tiene como objetivo cumplir con las demandas de la ejecución de una sola aplicación, experimenta una significativa interferencia entre aplicaciones cuando varias de estas comparten la GPU simultaneamente. Esta interferencia se produce por una

alta incidencia de fallos en el *Translation Lookaside Buffer* (TLB)[12], que es compartido entre todas las aplicaciones presentes en la GPU, causando latencias que afectan a cientos de hilos concurrentes y que eventualmente no pueden ser ocultadas por los hilos activos. Las mejoras sobre el TLB propuestas por *Ausavarungnirum. et al*[12] en base a este análisis, logran un incremento del 57.8 % en STP.

De la ejecución concurrente de aplicaciones pueden también emerger problemas de asignación de memoria. Estos problemas se manifiestan cuando a una aplicación le es denegada la asignación de un espacio de memoria en la GPU por no haber suficiente espacio disponible. Para paliar los riesgos de que esto suceda *Suzuki. et al* [13] propone *Mobile CUDA*, un sistema transparente que captura las llamadas al API de CUDA con la finalidad de que las aplicaciones puedan ser asignadas, suspendidas e incluso migradas entre las GPU del sistema. Mediante la evaluación de prestaciones de Mobile CUDA, se observa que este sistema reduce el tiempo de ejecución total un 18.4 % y genera un ahorro de energía del 5.5 %.

A diferencia de los trabajos previos mencionados, en este trabajo nos centramos en interferencias dentro de la jerarquía de memoria de la GPU, en particular en la LLC, que provocan un impacto en las prestaciones y proponemos un algoritmo de planificación que trata de minimizar estas interferencias.

2.2.2 Planificación teniendo en cuenta los recursos compartidos

Existe una gran cantidad de trabajos previos en la temática de investigación sobre la planificación de aplicaciones teniendo en cuenta los recursos compartidos, pero para aplicaciones CPU, por lo que no se encuentran directamente relacionados con este trabajo. Algunos de los más recientes se discuten a continuación.

Hay trabajos que caracterizan los procesos obteniendo información sobre estos para determinar que utilización del ancho de banda es la mas adecuada. Es el caso de [14], donde Xu et al. computan un *Ideal Average Bandwidth (IABW)* a partir de la utilización promedio de los anchos de banda para cada aplicación cuando estas se encuentran corriendo en solitario.

En un trabajo posterior, Feliu et al. [15] utiliza el *Online Average Transaction Rate (OATR)* como ancho de banda objetivo de cada quantum. La métrica pretende distribuir equitativamente el consumo promedio del ancho de banda a lo largo del tiempo de ejecución de las cargas. Una diferencia a tener en cuenta entre el IABW y el OATR, es que este último estima la utilización del ancho de banda en tiempo de ejecución, no requiriendo así de información

obtenida en ejecución individual.

Finalmente, en otro trabajo de Feliu et al. [16] se afirma que la degradación de prestaciones en ejecución concurrente depende principalmente de la contención causada por los concurrentes de aquellos recursos de los que el proceso requiera (ancho de banda de la jerarquía de memoria). Como consecuencia, algunas aplicaciones pueden ver afectado su progreso más que otras, produciéndose una desigualdad que puede tener efectos nocivos en las prestaciones globales.

Capítulo 3

Estructura y Organización del Planificador Propuesto

El presente capítulo presenta el entorno de planificación desarrollado En primer lugar, se introduce el diseño del entorno, destacándose los principales componentes del mismo y especificando tanto su finalidad como sus detalles de implementación. Posteriormente, se presenta un ejemplo del funcionamiento del entorno mediante una traza. Finalmente, se describen los *scripts* auxiliares que agilizan el lanzamiento de experimentos y el procesado de los resultados.

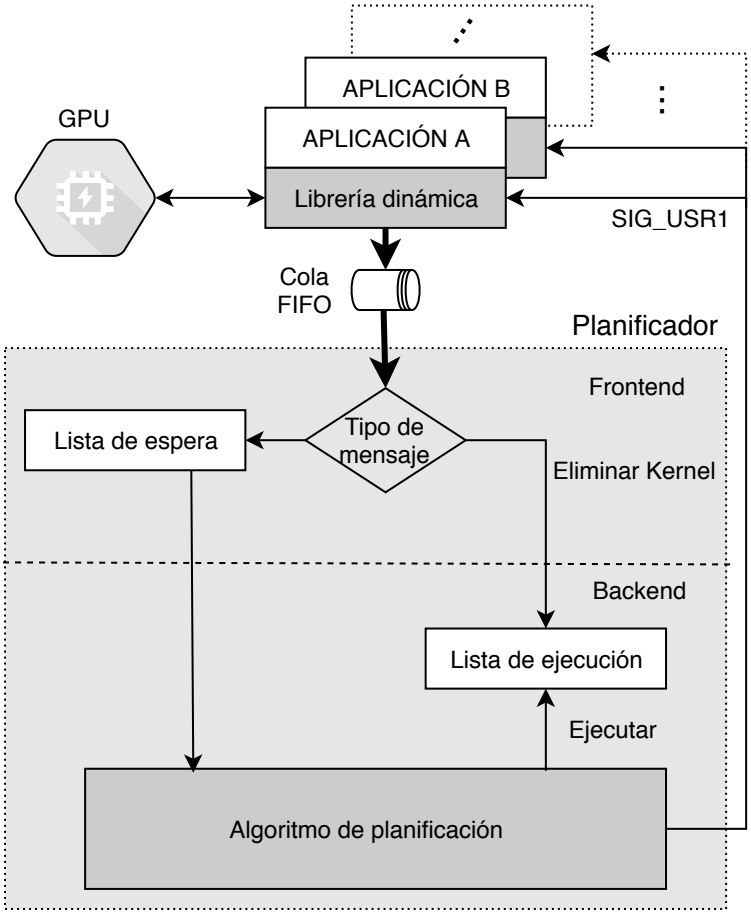


Figura 3.1: Diagrama de bloques del entorno de planificación desarrollado.

3.1 Diseño del entorno de planificación

Con la finalidad de mejorar las prestaciones e incrementar la utilización de la GPU se ha desarrollado un entorno software para la planificación de cargas multiprograma en GPU. Este entorno, desarrollado en C++11 [17] consta de tres componentes principales, los cuales se muestran en la figura 3.1 marcadas en gris oscuro:

- **Librería dinámica**. El objetivo de esta librería es capturar llamadas al Interfaz de Programación de Aplicaciones (API) de CUDA e inyectar el código necesario para soportar la funcionalidad del planificador.
- ***Frontend* del planificador**. Se trata del hilo de ejecución que inicializa el planificador, comunica las aplicaciones con el *backend* del mismo

y accede en su caso, a la información de *profiling* necesaria para la planificación.

- ***Backend* del planificador**. Consiste en un hilo que ejecuta el algoritmo de planificación, llevando a cabo las decisiones tomadas por éste.

Con tal de poder hacer uso del entorno, es necesario compilar los fuentes de las aplicaciones con las opciones del compilador *–shared* o *-lcudart*. Estas opciones permiten que sea posible capturar las llamadas al API de CUDA. Por otro lado, para que las aplicaciones se puedan ejecutar concurrentemente, es necesario lanzar el demonio de *Multi Process Service* (MPS). Además, para evitar falsas serializaciones de *kernels*, es recomendable definir el *compute mode* como *EXCLUSIVE_PROCESS* mediante la herramienta *nvidia-smi* de modo que las aplicaciones solo sean capaces de lanzarlos a través del servicio MPS activo y nunca de manera independiente.

A continuación, se describe la funcionalidad de cada uno de los componentes principales y se presenta un breve ejemplo de funcionamiento del entorno.

3.1.1 Librería dinámica

Como se ha comentado anteriormente, la librería dinámica intercepta llamadas específicas de las aplicaciones en ejecución al API de CUDA. Cuando una aplicación se dispone a lanzar un kernel, esta librería bloquea su ejecución hasta que el lanzamiento sea confirmado por el backend del planificador. La librería también crea un hilo (no mostrado en el diagrama) que espera a la finalización del kernel para notificarlo al frontend a través de la cola FIFO.

La librería se basa en la publicada en el repositorio *cudahook* [18], el cual implementa una librería dinámica que permite capturar los datos de los kernels a lanzar y mostrarlos por la salida estándar. Esta funcionalidad ha sido extendida para comunicar y sincronizar aplicaciones con el planificador, por tanto, mientras la librería original únicamente muestra por pantalla información sobre los kernels de la aplicación ejecutada, las modificaciones realizadas en este trabajo incluyen las características que se mencionan a continuación.

En el momento en el que se ejecutan las aplicaciones la variable de entorno *LD_PRELOAD* está definida de manera que las funciones definidas en la librería sustituyen a las nativas del API de CUDA. Las funciones interceptadas son las relacionadas con el registro y lanzamiento de kernels.

Con respecto al registro, antes de lanzar un kernel, la aplicación que lo lanzará debe registrarlo mediante la llamada `__cudaRegisterFunction`.

Esta llamada es interceptada y los parámetros de la llamada son almacenados en memoria para utilizarlos como identificador del kernel. Posteriormente, cuando la llamada a la función encargada de lanzar el kernel `cudaLaunch` es capturada, la librería se comunica con el frontend del planificador encolando un mensaje en una cola FIFO destinada a la comunicación interproceso, indicando en él el PID de la aplicación y el identificador del kernel que se desea lanzar.

La comunicación con el frontend es bloqueante, por lo que el lanzamiento del kernel por parte de la aplicación queda a la espera de confirmación, la cual llegará eventualmente desde el backend mediante la señal del sistema operativo *SIG_USR1*, para la cual la librería implementa un manejador. Una vez llega la confirmación, el kernel se lanza de forma asíncrona junto con el hilo encargado de monitorizar su estado. Cuando el kernel finaliza, este hilo informa al frontend de este evento mediante un mensaje que incluye el PID de la aplicación junto a la duración que ha tomado la ejecución del kernel.

3.1.2 Frontend del planificador

La decisión de separar el planificador en dos hilos ha sido tomada para evitar serialización entre la comunicación de las aplicaciones con el planificador y la planificación en sí.

El frontend del planificador es uno de los dos hilos de ejecución que conforman el planificador propuesto. El frontend se encarga de la inicialización (*main*) del planificador y de manejar la comunicación de las aplicaciones con el backend.

El frontend también se encarga de obtener información de caracterización de los kernels necesaria para las políticas de planificación. Esta información es recabada previamente mediante *profiling*. Para ello, el frontend toma como parámetro opcional un fichero (por defecto *kernel.db*) con los siguientes datos de los kernels a ejecutar:

- Identificador del kernel.
- Tamaño de grid.
- Tamaño de bloque.
- Ocupación de la memoria compartida.
- Ocupación del banco de registros.
- Consumo de ancho de banda en la cache L2.

Nótese que el formato del fichero kernel.db es extensible y permite incluir más datos además de los que se mencionan en este proyecto. Esta característica es útil para añadir nuevos algoritmos de planificación en futuros trabajos.

Al principio de su ejecución, el frontend inicializa una cola FIFO para comunicarse con las aplicaciones , lanza el backend y queda a la espera de recibir mensajes de las aplicaciones (generados por la librería descrita en la Sección 3.1.1) a través de la cola. El frontend puede recibir dos tipos de mensajes. El primer tipo se usa para indicar que la aplicación pretende lanzar un kernel, mientras que el segundo tipo indica que un kernel lanzado por una aplicación ha finalizado. Si se recibe un mensaje del primer tipo, el PID de la aplicación y el identificador del kernel se añaden a la lista de kernels en espera de ser ejecutados (lista de espera). Eventualmente, los kernels de la lista de espera serán transferidos a la lista de kernels en ejecución (lista de ejecución) por el backend como se detalla a continuación. Por otro lado, cuando se recibe un mensaje del segundo tipo indicando que un kernel ha finalizado, este es eliminado de la lista de ejecución.

3.1.3 Backend del planificador

El tercer componente del entorno es el hilo que realiza la planificación propiamente dicha. Este hilo espera activamente a que haya kernels en la lista de espera para ser lanzados, selecciona aquellos que deben hacerlo según el algoritmo de planificación. A continuación notifica a las aplicaciones correspondiente para permitir que realicen su lanzamiento.

El backend se configura con el algoritmo de planificación y el número máximo de kernels simultáneos que se permite ejecutar concurrentemente en la GPU. El algoritmo de planificación se ejecuta siempre que haya algún kernel en la lista de espera y el número de kernels ejecutándose en la GPU sea menor que el máximo permitido. En ese caso, el algoritmo de planificación selecciona los kernels de la lista de espera y los traslada a la lista de ejecución. Después, actualiza las estadísticas de utilización de la GPU y finalmente manda la señal *SIG_USR1* a las aplicaciones correspondientes para permitir el lanzamiento de los kernels seleccionados.

3.2 Ejemplo de funcionamiento

La figura 3.2 presenta un ejemplo de funcionamiento para un supuesto en el que existen tres aplicaciones constituidas únicamente por un kernel tratando de lanzarlos en una GPU con soporte para un máximo de dos kernels simultá-

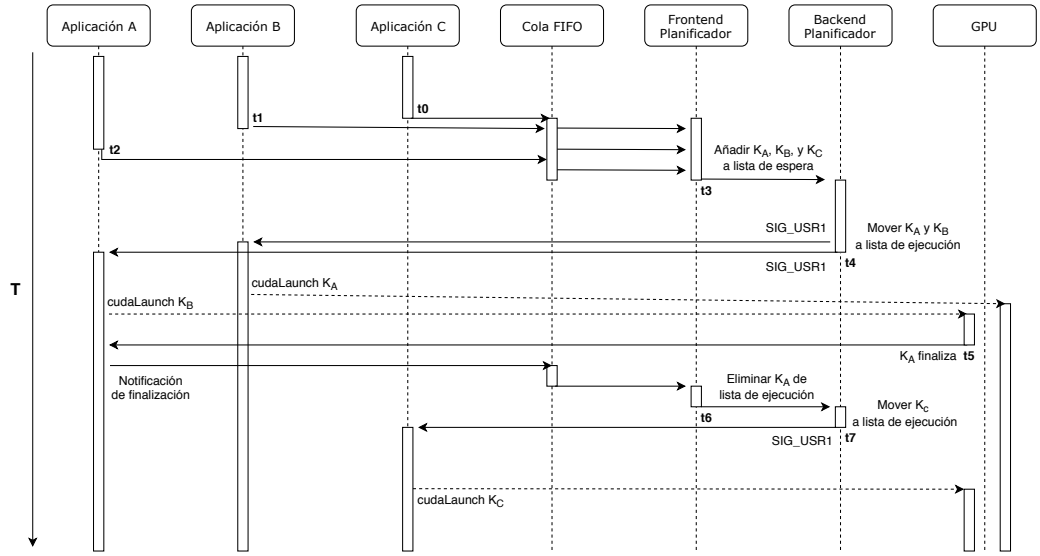


Figura 3.2: Ejemplo de planificación de tres aplicaciones para un máximo de 2 kernels concurrentes.

neos. Los rectángulos verticales representan actividad en el correspondiente componente del sistema mientras que las flechas horizontales representan las comunicaciones entre componentes.

En los instantes **t0**, **t1**, y **t2** las tres aplicaciones A, B, y C envían al frontend mediante la cola FIFO la información sobre los kernels que pretenden lanzar (K_A, K_B, y K_C). El frontend los desencola y añade a la lista de espera esta información en el instante **t3**. Posteriormente, en el instante **t4** el backend selecciona los kernels K_A y K_B, los añade a la lista de ejecución, y confirma su lanzamiento enviando la señal `SIG_USR1` a sus correspondientes aplicaciones, las cuales lanzan sus kernels a la GPU mediante la función `cudaLaunch`.

En el instante **t5** la ejecución del kernel de A finaliza, lo cual se reporta, mediante la cola FIFO al frontend, que en consecuencia en el instante **t6** retira K_A de la lista de ejecución. Finalmente, puesto que K_A ha finalizado su ejecución, el backend selecciona K_C en **t7**, añadiéndolo a la lista de ejecución y confirmando su lanzamiento a la aplicación C.

3.3 Scripts auxiliares

Para lanzar los experimentos en el planificador de manera automatizada, se han desarrollado una serie de scripts en *bash*, los cuales se describen a

continuación.

3.3.1 Script de inicialización para experimentos con el planificador

Este script inicializa la variable LD_PRELOAD con la librería descrita en la sección 3.1.1, prepara y realiza el lanzamiento del demonio de MPS y fija la frecuencia de reloj de la GPU. Finalmente, llama al script de control de los experimentos descrito en la sección 3.3.2.

3.3.2 Script de control de los experimentos

Inicialmente, el script de control de los experimentos genera un conjunto aleatorio de mezclas de aplicaciones. Para ejecutar el conjunto, el script de control construye dinámicamente un script que se usará para lanzar todas las mezclas.

Después, el script de control realiza un bucle donde, en cada iteración, inicia el planificador con una configuración distinta dependiendo del experimento, llama al script generado dinámicamente, espera a que finalize la ejecución de este último, detiene el planificador, y recopila los resultados, los cuales se almacenan en el mismo directorio donde se encuentra el script de control.

Nótese que es el script generado dinámicamente el que finalmente ejecuta cada una de las mezclas mediante diferentes llamadas al script de lanzamiento descrito en la sección 3.3.3. Además, el script generado dinámicamente también ejecuta una aplicación que procesa y almacena los resultados de la ejecución en un fichero correspondiente a la configuración correspondiente del planificador.

3.3.3 Script de lanzamiento

Mediante la llamada con parámetros a este script se pueden lanzar múltiples aplicaciones (es decir, una mezcla) de manera concurrente. El script interpreta los parámetros, lanzando las aplicaciones indicadas, las cuales generan resultados relevantes para la evaluación de prestaciones. Estos resultados se almacenan con formato *comma separated value* (csv), lo que facilita su análisis en aplicaciones de hoja de cálculo como por ejemplo *Libreoffice Calc o Google Sheets*.

Cada fichero corresponde a una configuración del planificador, cada línea representa el resultado de un lanzamiento de una mezcla, con el tiempo total de ejecución de la misma y las métricas STP y ANTT (explicadas en la

sección 4.3). Por otro lado, también se recoge la duración total de todos los kernels de cada aplicación ejecutada.

Capítulo 4

Entorno Experimental

En este apartado se describe el entorno experimental utilizado en este trabajo. Consiste en una GPU NVIDIA Titan X, con una arquitectura Maxwell de segunda generación. Tanto la arquitectura como las especificaciones técnicas de la tarjeta se detallan en la primera sección del presente capítulo. En la siguiente sección se describen brevemente las aplicaciones o benchmark que van a ser utilizados en este trabajo así como sus parámetros de lanzamiento o las modificaciones realizadas sobre estos. Finalmente, se detallan las métricas utilizadas en este trabajo para la evaluación de las prestaciones.

4.1 NVIDIA GeForce GTX Titan X

4.1.1 Arquitectura

La GeForce GTX Titan X es la GPU utilizada en este trabajo. Esta GPU está implementada con la segunda generación de la arquitectura Maxwell de NVIDIA. Un *Streaming Multiprocessor* (SM) con arquitectura Maxwell consta de 4 unidades *Single Instruction Multiple Thread* (SIMT) que pueden manejar warps de hasta 32 threads, lo que permite ejecutar hasta 128 threads simultáneamente por SM. Sin embargo, un SM puede tener asignados recursos para la ejecución de hasta 64 warps (i.e. 2048 threads), cuya ejecución se multiplexa en el tiempo. La tabla 4.1 muestra un resumen de las capacidades de la arquitectura Maxwell de segunda generación [19].

Threads per Warp	32
Warps per SM	64
Threads per SM	2048
Blocks per SM	32
Warp Allocation Granularity	4
Max Thread Block Size	1024

Tabla 4.1: Capacidades de la arquitectura Maxwell de segunda generación.

La jerarquía de cache en esta arquitectura dispone de una cache de L2 común para datos e instrucciones y compartida por todos los SM. Por otro lado, cada SM implementa una estructura de memoria configurable para su uso como cache de L1 de datos o bien como cache de textura, una cache de L1 de instrucciones, y una memoria compartida local al SM. Finalmente, por cada una de las 4 unidades SIMT se puede también encontrar un banco de registros de 32 bits [20].

4.1.2 Especificaciones técnicas

La NVIDIA GeForce GTX Titan X es una GPU de la gama de productos GeForce, principalmente instalados en equipos para consumidores que hacen uso de aplicaciones con una gran necesidad de computación gráfica, como por ejemplo, las aplicaciones de entretenimiento digital. Sin embargo, la compatibilidad de esta GPU con CUDA, el número de SM que ofrece, y su capacidad de memoria principal la hacen un sistema experimental adecuado para este trabajo, pues su gran cantidad de recursos facilita el desarrollo de experimentos donde varios kernels los comparten en paralelo. La tabla 4.2 resume las características de los principales componentes de la arquitectura de esta tarjeta [21].

CUDA Cores	3072
CUDA Cores per SM	128
Streaming Multiprocessors	24
Base Clock (MHz)	1000
Memory Clock	7.0 Gbps
DRAM Main Memory	12 GB
L1/texture cache	48 KB
L2 cache	3MB

Tabla 4.2: Especificaciones de la tarjeta NVIDIA GeForce GTX Titan X .

4.2 Benchmarks

Se han evaluado mezclas formadas a partir de un conjunto de seis aplicaciones. Estas aplicaciones han sido seleccionadas porque realizan un consumo de recursos tal que permite su ejecución en paralelo dentro de la GPU [22]. En otras palabras, la utilización de los recursos (cantidad de hilos, registros y memoria) realizada por los kernels de las aplicaciones seleccionadas es inferior o muy muy inferior a la capacidad de la GPU. Esta infrautilización permite observar con mayor claridad los potenciales beneficios que tiene la ejecución concurrente de distintos procesos en las prestaciones globales del sistema.

Las aplicaciones han sido modificadas con el objetivo de que su duración en ejecución individual sea similar (alrededor de unos 13 segundos). Además, debido a las limitaciones de la tarjeta en lo que respecta al número de unidades de coma flotante de doble precisión, se han sustituido las operaciones con este tipo de datos por operaciones de coma flotante de precisión simple.

Barnes-Hutt simulation (BH)

Esta aplicación pertenece a la suite lonestarGPU2.0. Es una implementación CUDA del algoritmo de Barnes et al. [23] para el cálculo de la fuerza ejercida por la gravedad en una agrupación de cuerpos. La carga utilizada es de dieciséis mil cuerpos y ciento veinte pasos o iteraciones.

Quality threshold clustering (QTC)

Esta aplicación pertenece a la suite SHOC. Es una implementación en CUDA del algoritmo de agrupación de genes descrito por Heyer et al. en [24]. La carga utilizada es la correspondiente al conjunto de datos predeterminado que se genera al lanzar la aplicación con el parámetro "*–size 2*".

Heartwall (HW)

Tanto esta como las aplicaciones posteriores pertenecen a la suite Rodinia. En particular Heartwall implementa un algoritmo para identificar las paredes del corazón en un vídeo de imagen por ultrasonido [25]. Para ejecutar la aplicación se usa como carga un vídeo de ciento cuatro fotogramas proporcionado por la suite y se indica por parámetro que se analice la totalidad del mismo.

LavaMD (LAVA)

Esta aplicación mide las fuerzas ejercidas por una agrupación de partículas en un espacio amplio [26]. Para lanzarla se utiliza el parámetro "*-boxes1d 5*".

Debido a la escasa duración de su kernel principal, para obtener resultados significativos se ha modificado el código original de LAVA con el objetivo de que su kernel sea lanzado de manera secuencial 10000 veces.

Speckle reducing anisotropic diffusion (SRAD)

Esta aplicación se utiliza para la reducción de ruido en imágenes de ultrasonido [27]. La aplicación se lanza con los siguientes parámetros "*10000 0.5 502 458*".

Leukocyte tracking (LEUKO)

Esta aplicación cuenta el número de leucocitos en tiempo real [28]. La carga utilizada es un vídeo de 100 fotogramas proporcionado por la suite, indicándose por parámetro que se analice la totalidad del mismo.

4.3 Métricas de prestaciones

Para el estudio de caracterización y la evaluación de las prestaciones de los algoritmos de planificación se han evaluado diferentes métricas, descritas a continuación. Todas las métricas relacionadas con el tiempo de ejecución pueden considerar tanto el tiempo global de la aplicación como el tiempo de ejecución en la GPU. Debido al enfoque de este trabajo, nos centraremos en medir las prestaciones en la GPU.

4.3.1 Instrucciones por ciclo (IPC)

Se refiere al número de instrucciones que se han completado por ciclo. La métrica puede referirse a un kernel o a una aplicación completa si se calcula como el sumatorio de todas las instrucciones lanzadas por los kernels de la aplicación dividido por el total de ciclos consumidos por la GPU.

4.3.2 Tiempo de ejecución

Este dato se obtiene para cada aplicación mediante el sumatorio de la duración de sus kernels debido a que éstas deben ejecutarse necesariamente en serie y no hay posible solapamiento entre ellos.

4.3.3 Media geométrica de *Speedup* o aceleración

Para comparar el efecto sobre una carga multiprograma de dos planificadores, se puede obtener la media geométrica de la aceleración obtenida en cada aplicación de la carga. La formula para n aplicaciones es:

$$Speedup_{gm} = \left(\prod_{i=1}^{n} \frac{T.\ ejec.\ base_i}{T.\ ejec.\ propuesta_i} \right)^{1/n}. \tag{4.1}$$

4.3.4 *Individual Speedup (IS)*

Ésta métrica tiene como objetivo cuantificar el impacto que tiene sobre una aplicación su ejecución en concurrencia con otras aplicaciones. Para calcular esta métrica, se divide su tiempo de ejecución en concurrencia con otras aplicaciones (MP) entre su tiempo si se ejecutase sola en el sistema (SP). Así pues,

$$IS = \frac{MP}{SP}. \tag{4.2}$$

4.3.5 *System throughput* (STP)

Métrica propuesta por *Eyerman y Eeckout* [29] que representa el trabajo completado por unidad de tiempo. Se calcula mediante el sumatorio para cada aplicación de la inversa de su Individual Speedup. Así pues,

$$STP = \sum_{i=1}^{n} \frac{1}{IS_i}. \tag{4.3}$$

4.3.6 *Average normalized turnaround time* (ANTT)

Propuesta también por *Eyerman y Eeckout* [29] para evaluar el rendimiento de cargas multiprograma, el ANTT cuantifica la percepción que puede tener el usuario sobre el efecto en el tiempo de ejecución de las aplicaciones al compartir recursos de manera concurrente. Es la media aritmética del IS de todas las aplicaciones. De este modo, el ANTT de una aplicación se calcula como:

$$ANTT = \frac{1}{n}\sum_{i=1}^{n} IS_i. \tag{4.4}$$

Capítulo 5

Estudio de Caracterización

En este capítulo se caracterizan las aplicaciones bajo estudio. En primer lugar, se realiza una caracterización de los benchmarks en ejecución individual, observándose en los resultados una gran diversidad en el número de kernels, tiempos de ejecución y anchos de banda consumidos en L2 y memoria principal, entre otros factores. Una vez realizada esta caracterización, se procede a la caracterización de las aplicaciones en ejecución concurrente. Para ello, se hace uso de dos microbenchmarks, el primero permite generar interferencias controladas en los recursos de la jerarquía de memoria compartidos (L2 y memoria principal), mientras que el segundo interfiere en los recursos de cómputo de la GPU. Los resultados del estudio de caracterización en ejecución concurrente muestran un comportamiento no homogéneo entre aplicaciones con respecto al impacto en las prestaciones dependiendo del tipo de interferencia.

Benchmark	Kernel	Peso (%)	T_{ex} (ms)	#Inst (M)	IPC	#Hilos (K)	BW_{L2} (GBps)	BW_M (GBps)	%mem stalls
BH	ForceCalculat	96,6	10,78	197	18,31	30	50,19	1,25	64,79
	Summarization	1,6	0,18	0,8	4,49	18	23,37	2,37	23,18
	TreeBuilding	0,8	0,09	0,9	9,27	36	55,33	1,21	52,79
	SortKernel	0,6	0,07	0,6	8,22	9	68,4	1,05	65,02
	BoundingBoxK	0,2	0,024	0,1	5,57	36	13,43	2,86	56,24
	EJEC. COMPLETA		4013,26	71945	17,93	30	49,74	1,27	63,88
HW	kernel	100	12,27	289	23,62	13	344,86	10,2	63,55
	EJEC. COMPLETA		1276,18	30137	23,62	13	344,86	10,2	63,55
LAVA	kernel_GPU_	100	0,67	35	52,59	16	46,24	0,32	5,28
	EJEC. COMPLETA		13392,46	704307	52,59	16	46,24	0,32	5,28
LEUKO	IMGVF_Ker	100	0,06	3	43,46	11	20,94	0,33	11,27
	EJEC. COMPLETA		6373,1	277001	43,46	11	20,94	0,327	11,27
SRAD	srad	39,8	0,054	2	31,46	230	158,56	8,99	15,1
	reduce	33,1	0,02	0,5	21,43	230	38,83	34,63	46,46
	srad2	20,2	0,03	0,9	32,99	230	320,37	177,69	32,61
	prepare	6,9	0,01	0,2	17,27	230	211,37	0,56	45,98
	EJEC. COMPLETA		1365,14	37503	27,47	230	155,26	50,97	31,15
QTC	QTC_device	82,1	2,09	36	17,65	9	43,49	13,38	13,22
	trim_Ungro	13,4	0,34	0,03	0,1	64	0,75	0,42	25,56
	reduce_kerne	3	0,08	0,002	0,03	1	0,65	0,55	82
	update_kernel	1,5	0,04	0,004	0,09	64	0,82	0,81	73,52
	EJEC. COMPLETA		1133,23	16437	14,5	7	35,84	11,07	17,84

Tabla 5.1: Características de los benchmark y sus kernels en ejecución individual.

5.1 Caracterización de los benchmark en ejecución individual

Para la caracterización de aplicaciones se hace uso de la herramienta *NVIDIA Visual Profiler* (NVVP) incluida en el CUDA SDK 8 [4]. Esta herramienta puede usarse tanto en ejecuciones individuales como concurrentes, pero sólo en el primer caso se puede obtener información de los contadores de prestaciones. NVVP exporta ficheros de resultados que recogen la duración total de la ejecución de todos los kernels lanzados durante el experimento de caracterización junto con los valores de los contadores de prestaciones que se desee (y sea posible, atendiendo a la mencionada limitación) analizar.

Los resultados de esta caracterización pueden observarse en la tabla 5.1. La tabla muestra, para cada benchmark y kernel, el peso en porcentaje de tiempo de GPU del kernel en el benchmark, su tiempo de ejecución, número de instrucciones, IPC, número máximo de hilos de GPU ocupados simultáneamente, los consumos de ancho de banda en L2 y memoria principal de la tarjeta, y el porcentaje de ciclos de parada (*stalls*) debido a dependencias de memoria. Además, para la ejecución completa de cada aplicación, la tabla

también muestra: a) los totales acumulados de tiempo de ejecución y número de instrucciones de todos los kernels; y b) la media (ponderada por el peso) de IPC, número de hilos ocupados, consumo de ancho de banda y porcentaje de ciclos de parada. Un kernel determinado puede lanzarse múltiples veces dentro de una misma aplicación. De ahí que los valores acumulados para la ejecución completa sean mucho mayores que la suma de los valores individuales de cada kernel.

Se observa una gran variedad de tiempos de ejecución entre los diferentes kernels, desde kernels que requieren para ejecutarse más de 10 milisegundos, a kernels que se completan en decenas de microsegundos. Como cabe esperar, el tiempo de ejecución se corresponde con el número de instrucciones ejecutadas. Por otro lado, también se observa una variedad similar en el IPC, métrica que ponderada con el peso del kernel en una aplicación, tiene una relación directa con las prestaciones globales. Nótese que salvo en SRAD, todas las aplicaciones presentan un kernel predominante (con un peso mayor del 80 %). También existe una gran variedad en el número de hilos ocupados por cada kernel, lo que señala que existen oportunidades de compartir la GPU, ya que muchos kernels por sí solos no son capaces de utilizarla completamente.

Sin embargo, es necesario tener en cuenta la interferencia en el consumo de ancho de banda, lo que requiere el diseño de algoritmos de planificación que eviten interferencias que afecten a las prestaciones. Como se comprueba en las secciones siguientes, ambos anchos de banda, el de la cache L2 y el de memoria principal pueden ser un cuello de botella. En este trabajo nos centramos en diseñar un algoritmo de planificación que tenga en cuenta la interferencia en la cache L2, entre otras razones, porque las aplicaciones estudiadas y sus kernels presentan un consumo de ancho de banda más heterogéneo que potencialmente proporciona más oportunidades para la mejora de prestaciones mediante planificación.

Un aspecto importante es que un alto consumo de ancho de banda en la cache L2 puede implicar a su vez un alto consumo del espacio ocupado en este recurso. Desafortunadamente, no nos es posible, por limitaciones de las herramientas empleadas (*profiler*), medir el porcentaje de cache ocupado por los kernels. Sin embargo, debe considerarse que, en lo que respecta a las prestaciones de las GPU, a diferencia de los procesadores convencionales, la contención en el ancho de banda en las caches es mucho más importante que en el espacio de almacenamiento, debido al elevado paralelismo en el acceso a memoria (deben gestionarse en algunos casos centenares de accesos simultáneos) y la baja localidad temporal de los datos [30].

Finalmente, cabe destacar que el porcentaje de ciclos de parada debido a la espera de datos que deben de ser traídos de memoria se encuentra, en general, relacionado con el ancho de banda consumido en la cache L2 y la

Memoria Principal. Así, BH, que es la aplicación que más ancho de banda consume, es también la que mayor porcentaje de ciclos de parada de este tipo presenta mientras que LEUKO, la aplicación con menor consumo, también presenta un menor porcentaje de *stalls*. Sin embargo, la relación no es tan evidente en otras aplicaciones debido a la gran capacidad de las GPU para ocultar la latencia de memoria mediante la ejecución de un número masivo de hilos.

5.2 Caracterización de los benchmark en ejecución concurrente

Una vez estudiado el comportamiento individual de los benchmarks, en esta sección se caracteriza su comportamiento en ejecución concurrente. Para ello, se han implementado microbenchmarks que permiten, de manera controlada, generar contención por los recursos compartidos para poder observar el impacto de estas interferencias en las prestaciones.

5.2.1 Diseño de microbenchmarks

Con la finalidad de estudiar el impacto de las interferencias en el uso del ancho de banda de la cache L2, la memoria principal y las unidades de cómputo de la GPU sobre las prestaciones de las aplicaciones, se han desarrollado dos *microbenchmarks*. Estos microbenchmarks causan interferencias cuando se lanzan de manera concurrente con la aplicación bajo estudio. El nivel de interferencia se regula mediante la modificación de constantes del microbenchmark previa compilación. Los dos microbenchmarks desarrollados introducen interferencias en el uso de la cache L2, memoria principal, y otras unidades funcionales de los multiprocesadores.

Interferencias en la jerarquía de memoria

Tanto para inyectar tráfico en la cache L2 como en la memoria principal se utiliza el mismo microbenchmark. En el caso de inyectar tráfico en la cache de L2, cada hilo del microbenchmark accede a un bloque distinto en la cache L2. Esta estrategia permite maximizar el consumo de ancho de banda en L2. Por otro lado, cuando se pretende inyectar tráfico en memoria principal, el microbenchmark reserva un vector de enteros de un orden de magnitud mayor que el tamaño de L2. Este vector es particionado mediante entrelazado entre los hilos y cada una de las partes es accedida secuencialmente por el hilo correspondiente. De esta manera, se consigue causar fallos en L2 con el

Algorithm 1 Pseucódigo del microbenchmark para interferir en memoria principal.

```
1:  kernel(int *A)
2:  {
3:    threadId_warp = threadId%32;
4:    int starting_position =
5:      STRIDE*BLOCKSIZE*blockId + threadId*STRIDE;
6:    int thread_stride = STRIDE*BLOCKSIZE*GRIDSIZE;
7:
8:    for(int i = starting_position;i<SIZE;i+=thread_stride)
9:      if(threadId_warp < DESIRED_PREDICATION)
10:       int tmp = A[i];
11: }
```

subsiguiente acceso a memoria principal.En ambos casos, la intensidad de la inyección de tráfico puede regularse variando el número de hilos activos de cada warp.

Respecto a la configuración del kernel del microbenchmark, el número de multiprocesadores en la GeForce GTX Titan X es 24, por lo que se lanza este número de bloques de hilos. Internamente, la GPU asigna cada bloque a un multiprocesador distinto. Para permitir que las aplicaciones se ejecuten en paralelo con el microbenchmark, el tamaño de bloque se fija a 512 hilos, un 50 % de los hilos totales de ejecución ya que cada multiprocesador puede tener asignados hasta 1024 hilos. Finalmente, nótese que el número de hilos por cada warp es 32, dando la posibilidad de tener 32 niveles distintos de consumo de ancho de banda.

El algoritmo 1 muestra el pseudocódigo que ejecuta cada hilo individual en el caso de interferir en memoria principal. El bucle situado entre las lineas 8 y 10 es el encargado de generar los accesos a memoria. En este bucle, `starting_position`, `SIZE`, y `thread_stride` representan, respectivamente, el primer bloque accedido por el hilo, el tamaño del vector mencionado anteriormente, y el desplazamiento con el que cada hilo accede al vector. Dentro del bucle, cada hilo accederá o no dependiendo de si su identificador dentro del warp es menor o no que la constante `DESIRED_PREDICATION`, la cual regula el número de hilos, entre 1 y 32, que acceden a memoria por warp.

En el caso de que se pretenda interferir en el acceso a la cache de L2, el pseudocódigo es similar al anterior, diferenciándose en la línea 10 del algoritmo cada hilo accede al mismo bloque durante toda la ejecución.

Algorithm 2 Pseucódigo del microbenchmark para interferir en las unidades funcionales del multiprocesador.

```
1:  kernel(int *A)
2:  {
3:    while (true)
4:    {
5:      int a = 0;
6:      int b = 1;
7:
8:      for(int i = 0;i<DESIRED_COMPUTE;i++)
9:        asm("mov.s32 %1,%0;" : "=r"(b),"=r"(a));
10:
11:     if(threadId==0) {
12:       int64_t clkns = clock64()+DESIRED_WAIT;
13:       while(clock64()<clkns){}
14:     }
15:     barrier();
16:   }
17: }
```

Interferencias en los multiprocesadores

Para interferir en el uso de las unidades funcionales de los multiprocesadores, se ha desarrollado un microbenchmark que consta de dos partes. En la primera parte (lineas 8 y 9 del Algoritmo 2), todos los hilos de su kernel realizan repetidamente una operación en ensamblador que requiere de las unidades funcionales de enteros. Esta operación se repite tantas veces como lo determine la constante `DESIRED_COMPUTE`.

En la segunda parte (líneas 11 a 15), se introduce un tiempo de espera controlado por la constante `DESIRED_WAIT`. Para implementar esta espera, el hilo 0 de cada bloque realiza una espera activa mientras que todos los demás hilos quedan inactivos en una barrera. Sólo cuando el hilo 0 termina su espera activa y entra en la barrera, se puede reanudar la ejecución de la primera parte del microbenchmark. La alternancia de la ejecución entre ambas partes permite, modificando las constantes mencionadas, regular la cantidad de interferencia que se introduce en las unidades funcionales.

Para ocupar todos los multiprocesadores, de manera análoga al microbenchmark que interfiere con la jerarquía de memoria, se lanzan 24 bloques. El tamaño de bloque se ajusta también a 512 hilos para que la ocupación de la GPU de las aplicaciones que se ejecuten con uno u otro microbenchmark

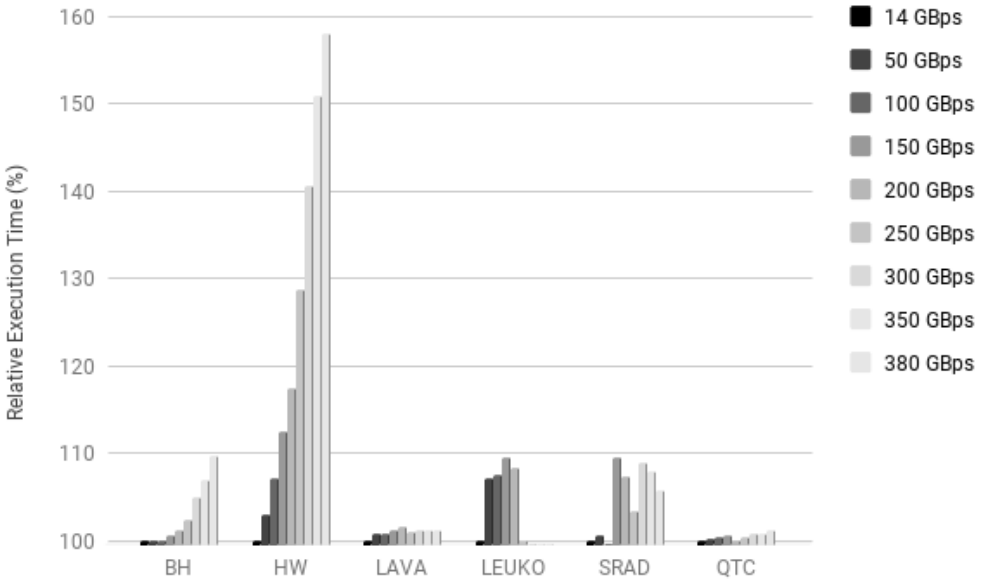


Figura 5.1: Impacto de las interferencias en la cache L2 en el tiempo de ejecución.

sea comparable.

5.2.2 Resultados

Impacto de las interferencias en la cache L2

Para estudiar el impacto en el tiempo de ejecución del nivel de interferencia en la cache L2, se ha ejecutado cada aplicación con diferentes consumos de ancho de banda en dicha cache por parte del microbenchmark. Debido a restricciones del sistema, no es posible interferir en la jerarquía de memoria sin ocupar hilos de cómputo en la GPU. Por lo tanto, para todos los experimentos en ejecución concurrente, se ha decidido que el microbenchmark ocupe un porcentaje fijo de hilos. Este porcentaje define los límites inferior y superior de consumo de ancho de banda del microbenchmark. En todos los experimentos en ejecución concurrente, como ya se ha indicado anteriormente, se ha establecido que el microbenchmark consuma un 50 % de los hilos de ejecución, permitiendo variar su consumo de ancho de banda en la cache L2 entre 14 y 380 GigaBytes por segundo (GBps).

La figura 5.1 muestra, variando el consumo de ancho de banda por parte del microbenchmark, el tiempo de ejecución relativo con respecto a un con-

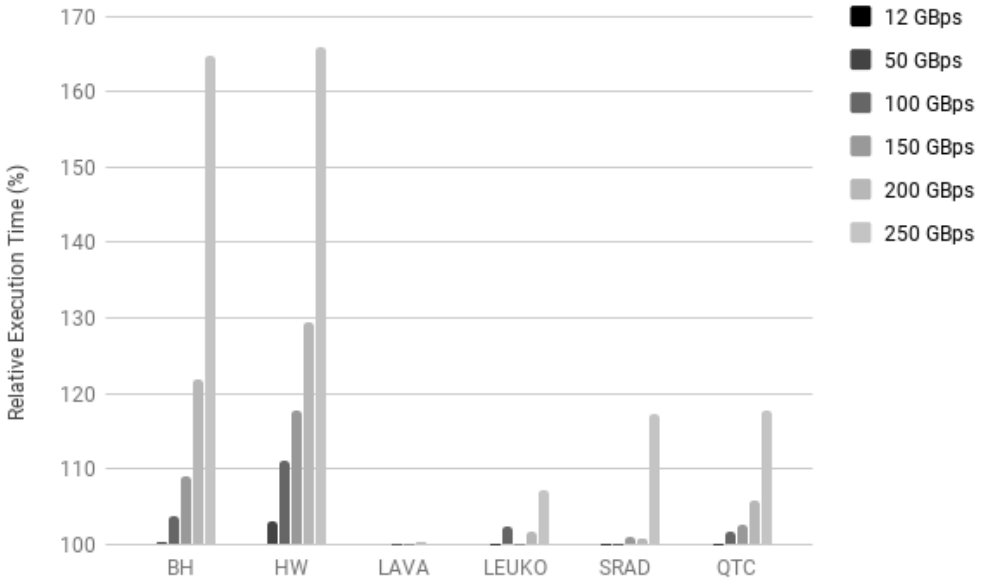


Figura 5.2: Impacto de las interferencias en memoria principal en el tiempo de ejecución.

sumo de 14 GBps, que es el mínimo posible para una ocupación del 50 % de la GPU. En general, la mayoría de las aplicaciones muestran una pérdida de prestaciones que se incrementa a medida que el microbenchmark consume más ancho de banda, aunque el impacto es diferente en cada aplicación.

Atendiendo al impacto en el tiempo de ejecución, se pueden clasificar las aplicaciones en tres tipos: a) benchmarks apenas afectados como LAVA o QTC, con incrementos del tiempo de ejecución de alrededor del 1 %; b) benchmarks medianamente afectados, como BH, LEUKO, y SRAD, con incrementos del tiempo de ejecución que llegan hasta alrededor de un 10 %; y c) benchmarks significativamente afectados como HW, con incrementos del tiempo de ejecución que pueden llegar hasta casi el 60 %. El motivo de esta penalización sobre HW es que cuando se ejecuta en solitario requiere un gran un gran consumo de ancho de banda en la cache L2 (ver tabla 5.1), lo que explica que sea la aplicación más afectada de entre todas las estudiadas.

Impacto de las interferencias en memoria principal

En la figura 5.2 se observa el resultado de un experimento análogo, pero esta vez consumiendo ancho de banda de memoria principal. Los resultados se muestran relativos a un consumo del microbenchmark de 12GBps, que

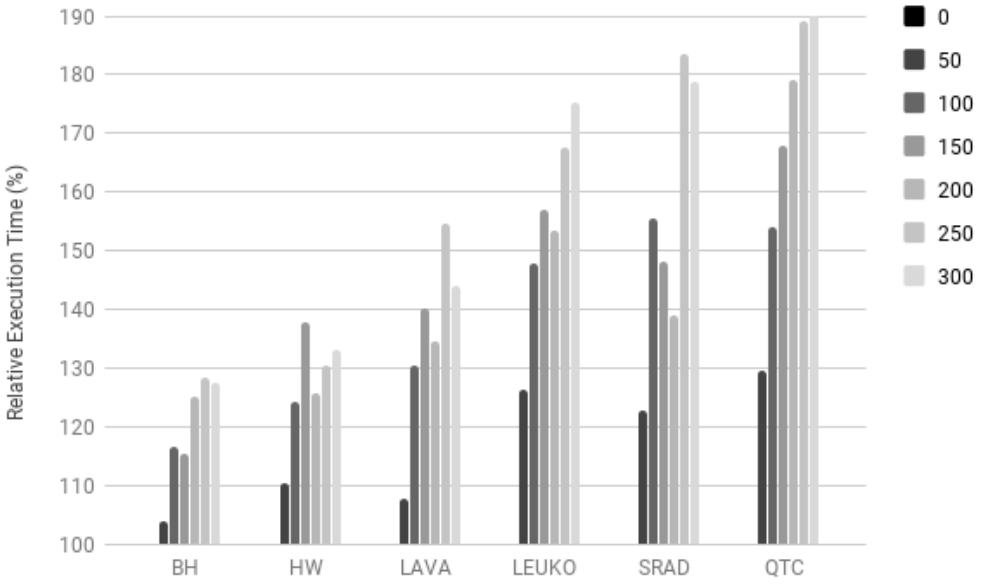


Figura 5.3: Impacto de las interferencias en los multiprocesadores en el tiempo de ejecución.

es el mínimo posible para una ocupación del 50 % de la GPU. En este caso nuestros experimentos demuestran que el microbenchmark puede alcanzar en memoria principal un consumo máximo de 250GBps.

Los benchmarks más afectados por estas interferencias son BH y HW, con incrementos en el tiempo de ejecución que pueden llegar a ser casi del 70 %, mientras que las demás aplicaciones presentan incrementos menores del 20 % en casi todos los casos. Sólo en el caso de máximo consumo de ancho de banda, SRAD y QTC ven sus prestaciones seriamente afectadas.

En el caso de las interferencias en memoria, no se observa una correspondencia directa entre el consumo de ancho de banda de memoria en solitario y la pérdida de prestaciones. Una causa posible de esta pérdida puede estar en el incremento de la tasa de fallos en la cache L2, lo que incrementaría la latencia de acceso a memoria y afectaría incluso a aplicaciones donde el cuello de botella de las prestaciones no resida en el ancho de banda de memoria. Lamentablemente, esta tesis no se puede comprobar directamente en la plataforma experimental, ya que no se permite medir el valor de los contadores de prestaciones en ejecución concurrente.

Sin embargo, si que se observa una relación directa de la pérdida de prestaciones con los porcentajes de ciclos de parada debido a dependencias de memoria mostrados en la caracterización de la ejecución individual (tabla

5.1). Por ejemplo, las aplicaciones con mayor porcentaje de ciclos de parada (BH y HW) son aquellas que sufren un impacto mayor en sus prestaciones por las interferencias en memoria principal, mientras que LAVA y LEUKO, que presentan un porcentaje mucho menor, también son menos sensibles a este tipo de interferencias.

Impacto de las interferencias en los multiprocesadores

Para los experimentos realizados con el microbenchmark que causa interferencias en el uso de los multiprocesadores se ha fijado el valor del parámetro `DESIRED_WAIT` a 5000 nanosegundos y se ha variado el valor de `DESIRED_COMPUTE` entre 0 y 300 iteraciones. La figura 5.3 muestra los resultados. Aunque todas las aplicaciones se ven afectadas con la misma tendencia por la contención en el uso de los recursos de cómputo, la magnitud de esta tendencia varía entre aplicaciones. Las aplicaciones que menos afectadas se ven son BH y HW, siendo el resto afectadas de una manera más notable llegando en casos como LEUKO, QTC y SRAD a incrementar su tiempo de ejecución en más de un 80 %.

Capítulo 6

Algoritmo de Planificación SHARK

En este capítulo se presenta nuestra propuesta de algoritmo de planificación SHARK. La primera sección describe el algoritmo, mientras que en la segunda sección se realiza la evaluación experimental, comenzando por el speedup con respecto al planificador por defecto incluido en MPS y siguiendo por un análisis de las métricas STP y ANTT.

6.1 Descripción de SHARK

En esta sección se describe SHARK, un algoritmo para la planificación de cargas GPU multiprograma. SHARK se basa en los resultados del capítulo 5, donde se observa que las aplicaciones presentan un comportamiento heterogéneo en lo que respecta a su consumo de ancho de banda y al impacto en sus prestaciones dependiendo del ancho de banda disponible (es decir, consumido por otras aplicaciones ejecutándose en la misma GPU). Este hecho ofrece oportunidades para el diseño de estrategias de planificación que minimicen la interferencia en la jerarquía y mejoren las prestaciones.

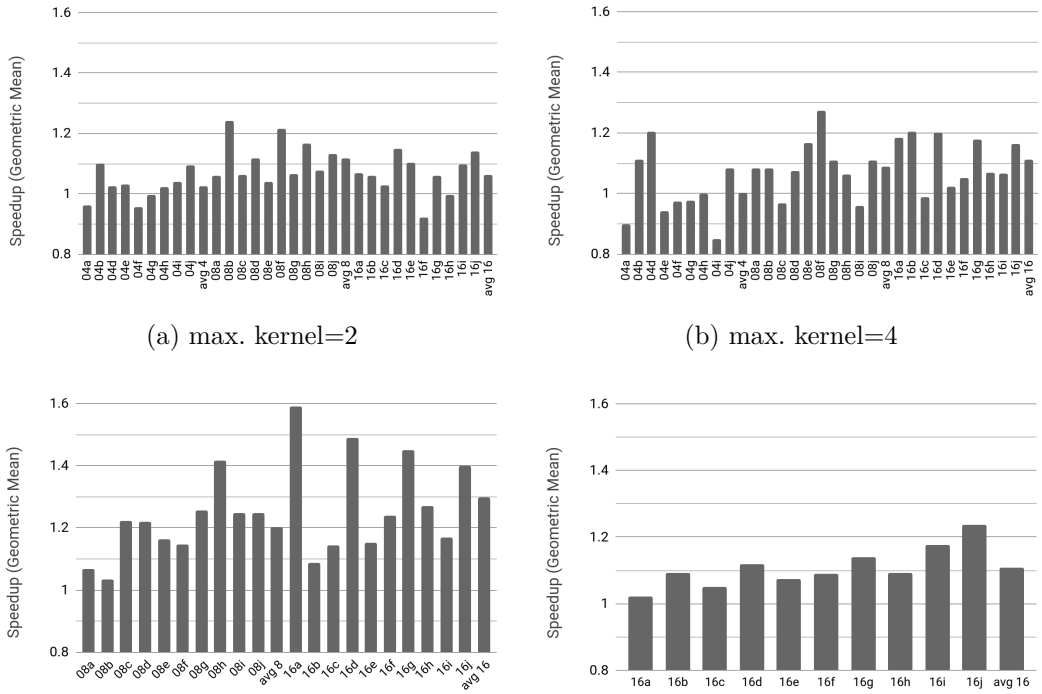
Para evitar interferencias que afecten a las prestaciones, SHARK evita que los kernels con un gran consumo de ancho de banda en cache L2 se ejecuten al mismo tiempo. Para ello, mantiene una lista ordenada $\{K_0 K_1 ... K_{N-1}\}$ de los N kernels activos y se asegura de que en todo momento en la GPU sólo se ejecutan el kernel con mayor consumo de ancho de banda de L2 (K_0) junto con los M kernels de menor consumo ($K_{N-M} ... K_{N-1}$). El valor de M se establece teniendo en cuenta el parámetro de configuración del planificador que establece el número de kernels que se pueden ejecutar en paralelo.

En este trabajo, SHARK determina el consumo de ancho de banda de cada kernel a partir de información de *profiling* almacenada en el fichero `kernel.db`, mientras que la información sobre los recursos de cómputo requeridos puede obtenerse *online* en el momento en que se registra cada kernel. Es posible plantear una propuesta completamente dinámica que prediga el consumo de ancho de banda de cada kernel a partir de la última instancia ejecutada. Sin embargo, en nuestra plataforma hardware esta propuesta no es implementable ya que la GPU no permite obtener el valor de los contadores de prestaciones en ejecución concurrente.

6.2 Evaluación experimental

En esta sección SHARK se evalúa y compara con el planificador por defecto incluido en MPS. Para ello, se ha implementado un algoritmo de planificación *baseline* que delega a MPS la decisión de si un kernel debe ejecutarse o no a medida que llegan las peticiones de lanzamiento al planificador. La única restricción que impone este algoritmo es el número máximo de kernels que se permite que se ejecuten en paralelo en la GPU. Tanto para SHARK como para MPS, este límite se ha variado en los experimentos que se presentan en esta sección entre los valores 2, 4, 8, y 16.

Para la evaluación de los algoritmos de planificación se han generado tres conjuntos con 10 mezclas de aplicaciones cada uno. En el primer conjunto las



(a) max. kernel=2

(b) max. kernel=4

(c) max. kernel=8

(d) max. kernel=16

Figura 6.1: Speedup de SHARK con respecto a MPS en la GPU. Para cada mezcla, el valor de la columna muestra la media geométrica del speedup de las aplicaciones.

mezclas están compuestas por 4 aplicaciones aleatorias, en el segundo por 8 aplicaciones, y en el tercero por 16, todas ellas obtenidas de manera aleatoria. Para cada experimento con un limite máximo de kernels en ejecución, se han evaluado sólo aquellas mezclas compuestas por un número de aplicaciones mayor o igual que ese límite. Por ejemplo, para los experimentos con un límite máximo igual a 8, se han evaluado las mezclas compuestas por 8 y 16 aplicaciones. Esto es debido a que un kernel de una aplicación sólo se lanza si los kernels anteriores de la misma aplicación han finalizado, lo que impide que se pueda alcanzar el limite máximo de kernels en ejecución cuando se usan mezclas cuyo número de aplicaciones es menor que dicho límite.

6.2.1 Speedup

La figura 6.1 muestra, para 4 límites máximos de kernels en paralelo y para cada mezcla, la media geométrica del speedup en la GPU de SHARK con

respecto a MPS de las aplicaciones. Cada columna representa el valor de la media geométrica de una mezcla. Las mezclas se nombran siguiendo el formato "`XXy`", donde los dígitos "`XX`" representan el número de aplicaciones contenidas en la mezcla, y la letra "`y`" identifica a la mezcla dentro del conjunto de mezclas con el mismo número de aplicaciones. Además, se han añadido columnas (`avg`) con el promedio de speedup para cada conjunto de mezclas.

Se puede observar que SHARK mejora los resultados de MPS en la mayoría de las mezclas. Los mejores resultados para SHARK se dan con un límite de kernels en paralelo igual a 8, donde SHARK mejora a MPS en todas las mezclas, llegando a aceleraciones superiores al 40 % en algunos casos. Hemos comprobado que estas grandes aceleraciones se deben principalmente al número de kernels en la mezcla con un gran consumo y sensibilidad a la contención en la cache L2. Por ejemplo, las mezclas `16a`, `16d`, `16g` y `16j`, las cuales tienen en común que contienen 4 instancias de la aplicación HW, presentan aceleraciones superiores a 1,4 cuando el límite de kernels es igual a 8. Nótese que HW es la aplicación cuyos kernels consumen más ancho de banda de L2 (véase tabla 5.1) y a la vez es una de las más sensibles a la contención en la jerarquía de memoria, como se discutió en la sección 5.2.2.

Casi todos los conjuntos de mezclas obtienen un speedup en promedio mayor que 1. Sólo en el caso del conjunto de mezclas de 4 aplicaciones con un límite de kernels en paralelo igual a 4 se observa una ligera ventaja de MPS. En general, cuando el tamaño de la mezcla coincide con el límite de kernels, la ventaja de SHARK es menor. Esto se debe a que en este caso la única diferencia entre SHARK y MPS es el orden de lanzamiento de los kernels. Es decir, SHARK no tiene margen para decidir qué kernels de entre los activos se ejecutan en un momento determinado, lo que limita sus potenciales mejoras. Esto explica también porqué SHARK ofrece peores resultados cuando el límite de kernels pasa de 8 a 16, ya que en este último caso sólo se han evaluado mezclas de 16 aplicaciones. Lamentablemente, las limitaciones de nuestra plataforma impiden evaluar mezclas de 32 aplicaciones.

En aquellos casos en los que las prestaciones de SHARK no mejoran a las de MPS, la degradación de prestaciones es relativamente pequeña (sólo hay un caso donde el speedup es menor que 0,9). Estos casos se explican porque el speedup se calcula como una media geométrica, lo que magnifica situaciones donde una de las aplicaciones de la mezcla es repetidamente penalizada por el algoritmo al ser sus kernels siempre los últimos seleccionados para su lanzamiento. Esto implica que el algoritmo tiene todavía cierto margen de mejora y debería refinarse para considerar la equidad (*fairness*) en la planificación, lo que se plantea como trabajo futuro.

La figura 6.2 muestra la media de speedup para cada conjunto de mezclas.

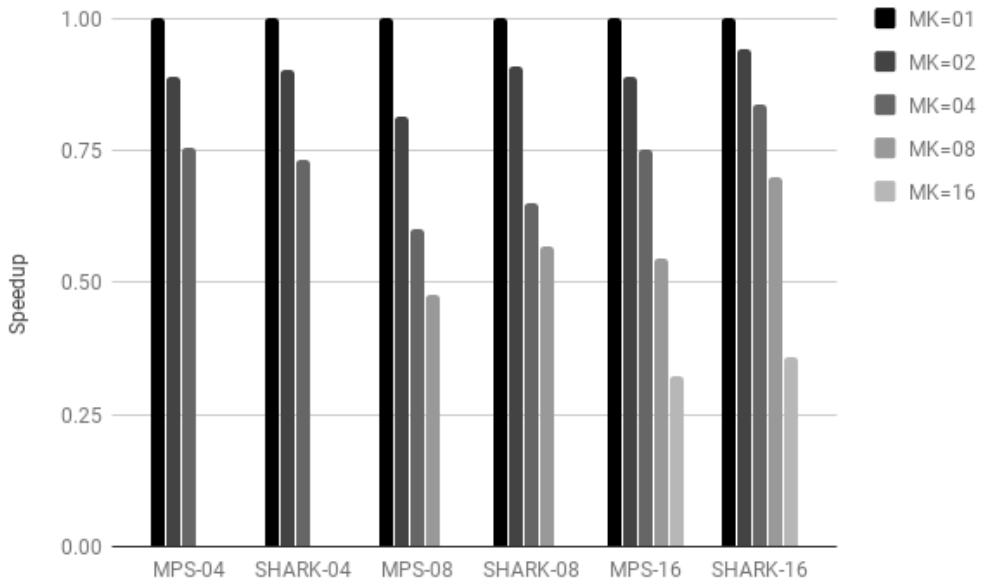


Figura 6.2: Media geométrica del Speedup para los diferentes conjuntos de mezclas de los algoritmos `SHARK` y `MPS` con respecto a la ejecución en solitario variando el límite máximo de kernels en la GPU.

Sin embargo, a diferencia de la figura 6.1, en esta figura todos los speedups se calculan relativos al tiempo de ejecución de cada aplicación cuando se ejecuta en solitario (es decir, max. kernel igual a 1 o MK igual a 1). Cada grupo de columnas muestra los resultados promedio de un algoritmo de planificación (MPS o SHARK) para un conjunto de mezclas (mezclas de 4, 8 o 16 aplicaciones).

Se observa que la aceleración es menor que 1, es decir, hay una deceleración y se reduce a medida que se incrementa el límite máximo de kernels en la GPU. Este resultado es normal, ya que al aumentar el número de kernels en paralelo las aplicaciones cuentan con menos recursos en la GPU, lo que incrementa el tiempo de ejecución de los kernels. Este efecto es más evidente en los conjuntos de mezclas con mayor número de aplicaciones. Sin embargo, en casi todos los casos, SHARK ofrece una menor reducción de prestaciones que MPS a medida que se incrementa el límite máximo. En el caso de las configuraciones con un límite de kernels en la GPU igual a 8 es donde mejor se aprecia este efecto, corroborando que SHARK obtenga sus mejores resultados para ese límite en la figura 6.1.

En resumen, como cabe esperar, los resultados muestran que a medida que se aumenta el número de kernels ejecutándose en paralelo en la GPU, la

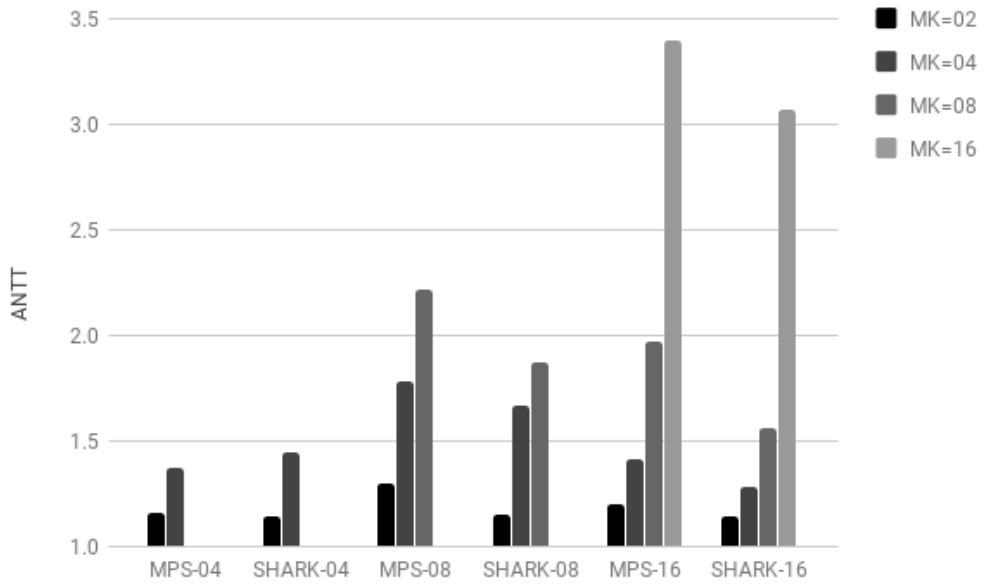


Figura 6.3: Valor medio de ANTT para los diferentes conjuntos de mezclas de los algoritmos `SHARK` y `MPS` variando el límite máximo de kernel en la GPU.

contención aumenta y por tanto las prestaciones individuales de las aplicaciones se reducen respecto a la ejecución en solitario. Sin embargo, SHARK consigue paliar este efecto mejorando las prestaciones con respecto al planificador base de la GPU.

6.2.2 STP y ANTT

Como se ha comentado anteriormente, las métricas STP y ANTT permiten evaluar los algoritmos de planificación con respecto a la productividad del sistema y el efecto de la concurrencia en el tiempo de ejecución, respectivamente. La figura 6.3 muestra el valor medio de ANTT para cada conjunto de mezclas de los algoritmos estudiados variando el límite máximo de kernels en la GPU.

La figura corrobora el aumento del tiempo de ejecución de la carga con el número de aplicaciones y el nivel de paralelismo que se permite en la GPU, llegando la ejecución a enlentecerse hasta más del triple en mezclas compuestas por 16 aplicaciones cuando se permite el máximo nivel de paralelismo. Como se puede apreciar, SHARK proporciona mayores beneficios en el ANTT con cargas relativamente grandes (8 o 16 aplicaciones) y altos niveles de paralelismo (límite máximo de kernels en la GPU de 8 o 16).

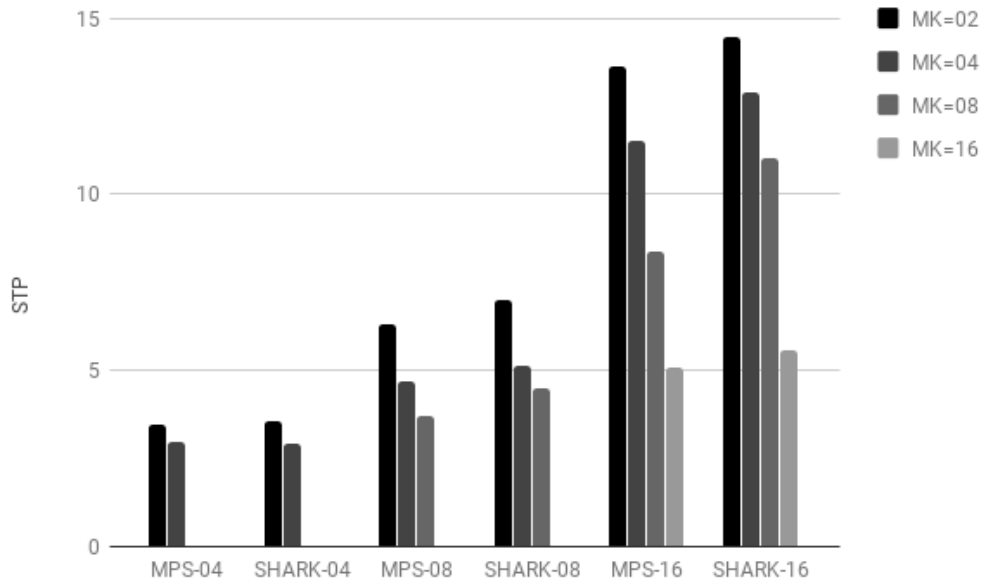


Figura 6.4: Valor medio de STP para los diferentes conjuntos de mezclas de los algoritmos `SHARK` y `MPS` variando el límite máximo de kernels en la GPU.

Con respecto a la productividad del sistema, la figura 6.4 muestra los resultados de STP promedio para cada conjunto de mezclas y configuración del planificador. Se observa que, independientemente del algoritmo de planificador utilizado, el valor de STP se resiente si se aumenta el nivel de paralelismo de la GPU y el tamaño de las mezclas. Esto demuestra que, en general, cuando se lanzan varias aplicaciones en la GPU se genera contención en el acceso a los recursos de cómputo y la jerarquía de memoria. También se observa que SHARK reduce los efectos de esta contención al presentar una productividad mayor. El impacto de SHARK sobre la productividad es más positivo en las mezclas de 16 aplicaciones, donde existe mayor contención.

En resumen, los resultados de ANTT y STP muestran que el tiempo de ejecución de las aplicaciones y la productividad del sistema se ven más negativamente afectados cuando se utilizan cargas multiprograma de gran tamaño en configuraciones del planificador donde la GPU es compartida por un alto número de aplicaciones. En estas situaciones, los beneficios de SHARK son más evidentes, lo cual hace que SHARK sea un algoritmo de planificación especialmente adecuado para futuros sistemas donde la GPU se comparte por un número cada vez mayor de aplicaciones.

Capítulo 7

Conclusiones y Trabajo Futuro

En este capítulo se resumen las principales conclusiones de este trabajo y se presentan algunas directrices sobre el trabajo futuro.

7.1 Conclusiones

El uso de las en GPU en el ámbito de la computación de altas prestaciones (HPC) sigue creciendo. Actualmente, alrededor de un 50 % de los 10 primeros computadores de la lista Top500 implementan dispositivos GPU. Por otro lado, el crecimiento de la potencia de cálculo de estos dispositivos representa una oportunidad para explotar la concurrencia en la ejecución de múltiples aplicaciones GPU o *kernels*. No obstante, en su estado actual, el soporte a la concurrencia todavía es limitado y las prestaciones de las aplicaciones con una gran dependencia de los recursos compartidos pueden verse seriamente afectadas. Por este motivo, se hace patente la necesidad de un planificador de aplicaciones que permita reducir las interferencias entre aplicaciones concurrentes, reduciendo de este modo el impacto sobre las prestaciones globales del sistema.

El presente trabajo ha realizado las siguientes aportaciones: a) una infraestructura de planificación de cargas multiprogramadas para GPU, b) un estudio de caracterización de aplicaciones GPGPU en ejecución individual y concurrente y c) una propuesta y evaluación de un nuevo algoritmo de planificación que tiene en cuenta el consumo de ancho de banda de L2 de las aplicaciones.

La infraestructura de planificación desarrollada permite controlar el número de kernels en ejecución en la GPU en cada momento, así como soportar diversos algoritmos de planificación. Hemos comprobado que el entorno de planificación funciona correctamente en una GPU Nvidia Titan X con arquitectura Maxwell de segunda generación.

Con respecto al estudio de caracterización, atendiendo a los resultados obtenidos se pueden extraer las siguientes conclusiones:

- Existe una diversidad considerable en las características generales entre aplicaciones e incluso entre los kernels de una misma aplicación, lo que permite que los algoritmos de planificación puedan explotar estas diferencias con el objetivo de maximizar las prestaciones.

- El consumo de ancho de banda de la cache L2 es más heterogéneo entre aplicaciones que el de Memoria Principal, por lo que un algoritmo de planificación inicial que explote las diferencias entre aplicaciones puede centrarse en el consumo de este recurso.

- Aquellas aplicaciones que se ven más afectadas por el consumo de ancho de banda, son las que menos se ven afectadas por la utilización de recursos de cómputo.

A partir del estudio de caracterización, se ha propuesto SHARK, un algoritmo de planificación que pretende evitar que más de un kernel con consumo elevado de ancho de banda de L2 concurra al mismo tiempo en la GPU, para así reducir el tiempo de ejecución de los kernels mediante la reducción de interferencias en el uso del ancho de banda. De la evaluación del algoritmo propuesto se extraen las siguientes conclusiones:

- Independientemente del algoritmo utilizado, el tiempo de ejecución de los kernels y la productividad del sistema se incrementa a medida que aumenta la contención en los recursos de la GPU, es decir, a medida que se aumenta el número de aplicaciones por carga multiprogramada y el número de kernels que se permite ejecutar en paralelo por parte del planificador.

- Al aumentar la contención en la GPU, SHARK presenta un comportamiento más escalable, el cual es más evidente en altos niveles de contención.

De lo anteriormente expuesto, se concluye que existen beneficios en el uso del planificador SHARK frente al planificador por defecto de MPS ya que SHARK permite paliar en parte los efectos nocivos de la contención en la jerarquía de memoria. Es por ello que consideramos que el algoritmo propuesto es adecuado para futuros sistemas donde siguiendo la tendencia actual, se espera que la GPU disponga de muchos más multiprocesadores y, por tanto, pueda ser compartida por un número cada vez mayor de aplicaciones independientes.

7.2 Trabajo Futuro

Los pasos a seguir tras este trabajo incluyen la mejora y el desarrollo de implementaciones que permitan mejorar todavía más las prestaciones del planificador. Estas mejoras se deben enfocar en añadir restricciones que prevengan que una gran confluencia de aplicaciones simultáneas causen problemas en el lanzamiento de las cargas así como considerar la equidad entre aplicaciones.

Además, pretendemos adaptar el planificador a GPU que permitan obtener los valores de los contadores de prestaciones en ejecución concurrente y así poder plantear un algoritmo de planificación enteramente *online*. Una vez desarrollado y evaluado este algoritmo, nuestra intención es publicar sus resultados en una conferencia y/o revista de primer nivel.

Finalmente, se continuará experimentando con distintas propuestas y modificaciones de los algoritmos con miras de encontrar una solución que permita minimizar la pérdida de prestaciones en contextos donde el nivel de concurrencia sea masivo.

Bibliografía

[1] NVIDIA Corporation. Product families, graphics cards, and technologies|nvidia. `http://www.nvidia.com/page/products.html`, June 2017. (Accessed on 07/03/2017).

[2] Enhua Wu and Youquan Liu. Emerging technology about gpgpu. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 618–622. IEEE, 2008.

[3] TOP500.org. Home | top500 supercomputer sites. `https://www.top500.org/`, June 2018. (Accessed on 01/09/2018).

[4] NVIDIA Corporation. Programming guide :: Cuda toolkit documentation. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, June 2017. (Accessed on 07/03/2017).

[5] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[6] NVIDIA Corporation. Programming guide :: Cuda toolkit documentation. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#concurrent-kernel-execution`, June 2017. (Accessed on 07/03/2017).

[7] NVIDIA Corporation. Kepler tuning guide :: Cuda toolkit documentation. http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html#hyper-q, June 2017. (Accessed on 07/03/2017).

[8] PETER MESSMER. Unleash legacy mpi codes with kepler's hyper-q | the official nvidia blog. https://blogs.nvidia.com/blog/2012/08/23/unleash-legacy-mpi-codes-with-keplers-hyper-q/, August 2012. (Accessed on 05/21/2018).

[9] NVIDIA Corporation. Multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, May 2015. (Accessed on 07/03/2017).

[10] Lingyuan Wang, Miaoqing Huang, and Tarek El-Ghazawi. Towards efficient gpu sharing on multicore processors. *SIGMETRICS Perform. Eval. Rev.*, 40(2):119–124, October 2012.

[11] Rommel A.Q. Cruz, Cristiana Bentes, Bernardo Breder, Eduardo Vasconcellos, Esteban Clua, Pablo M.C. de Carvalho, and Lúcia M.A. Drummond. Maximizing the gpu resource usage by reordering concurrent kernels submission. *Concurrency and Computation: Practice and Experience*, pages e4409–n/a. e4409 cpe.4409.

[12] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 503–518, New York, NY, USA, 2018. ACM.

[13] Taichiro Suzuki, Akira Nukada, and Satoshi Matsuoka. Efficient execution of multiple cuda applications using transparent suspend, resume and migration. In Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, pages 687–699, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[14] Di Xu, Chenggang Wu, and Pen-Chung Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 237–248, New York, NY, USA, 2010. ACM.

[15] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Addressing fairness in smt multicores with a progress-aware scheduler. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 187–196, May 2015.

[16] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Addressing fairness in smt multicores with a progress-aware scheduler. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 187–196, May 2015.

[17] ISO/IEC 14882:2014(E) Programming Language C++. Standard, International Organization for Standardization, Geneva, CH, 2014.

[18] Nathan Chong. nchong/cudahook: Intercepting cuda runtime calls with ld_preload. `https://github.com/nchong/cudahook`, March 2014. (Accessed on 06/12/2018).

[19] NVIDIA Corporation. Programming guide :: Cuda toolkit documentation. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-5-x`, June 2017. (Accessed on 07/03/2017).

[20] NVIDIA Corporation. Geforce gtx 980 whitepaper. `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF`, September 2014. (Accessed on 07/03/2017).

[21] NVIDIA Corporation. Geforce gtx titan x | specifications | geforce. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications`, June 2017. (Accessed on 07/03/2017).

[22] David Baselga-Masia. Caracterización de aplicaciones gpu y estudio de interferencias en una gpu nvidia geforce titan x. https://riunet.upv.es/handle/10251/86673, September 2017.

[23] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[24] Laurie J Heyer, Semyon Kruglyak, and Shibu Yooseph. Exploring expression data: identification and analysis of coexpressed genes. *Genome research*, 9(11):1106–1115, 1999.

[25] Lukasz G Szafaryn, Kevin Skadron, and Jeffrey J Saucerman. Experiences accelerating matlab systems biology applications. In *Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits*, pages 1–4, 2009.

[26] Lukasz G Szafaryn, Todd Gamblin, Bronis R De Supinski, and Kevin Skadron. Experiences with achieving portability across heterogeneous architectures. *Proceedings of WOLFHPC, in Conjunction with ICS, Tucson*, 2011.

[27] Yongjian Yu and Scott T Acton. Speckle reducing anisotropic diffusion. *IEEE Transactions on image processing*, 11(11):1260–1270, 2002.

[28] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating leukocyte tracking using cuda: A case study in leveraging manycore coprocessors. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009.

[29] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, May 2008.

[30] Francisco Candel, Salvador Petit, Alejandro Valero, and Julio Sahuquillo. Improving GPU Cache Hierarchy Performance with a Fetch and Replacement Cache. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 235–248, Cham, 2018. Springer International Publishing.