UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Caracterización de aplicaciones GPU y estudio de interferencias en una GPU Nvidia GeForce Titan X

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: David Baselga Masiá

Tutores: Salvador Petit Martí, Julio Sahuquillo Borrás

Curso 2016-2017

Resumen

Actualmente la implementación de las GPGPU intenta maximizar el uso de los recursos disponibles, pero todavía no contempla la compartición de recursos entre distintas aplicaciones. En este trabajo se exploran las posibilidades de paralelización que las GPGPU ofrecen con el objetivo de diseñar un planificador que tenga en cuenta la contención en los recursos compartidos y la simbiosis entre aplicaciones. Mediante el uso de las herramientas proporcionadas por NVIDIA se caracteriza el comportamiento de una serie de aplicaciones GPGPU en ejecución individual y concurrente. Los resultados de los experimentos muestran que existe una correlación entre las tasas de acceso a los recursos compartidos y la degradación de las prestaciones, lo que abre la puerta al diseño de planificadores inteligentes para GPGPU.

Palabras clave: NVIDIA GeForce GTX Titan X, Caracterización, GPGPU, Planificadores simbióticos, Contención en la jerarquía de memoria

Abstract

Nowadays GPGPUs maximize available resource usage, but they do not yet consider resource sharing among different GPGPU applications. In this work, GPGPU capabilities to execute several applications in parallel are explored in order to design a scheduler that takes into account resource sharing and application symbiosis. Using NVIDIA tools, the behavior of applications is characterized in both individual and parallel execution. Results show a correlation between shared resource access rates and IPC degradation, enabling the design of intelligent schedulers for GPGPUs.

Key words: NVIDIA GeForce GTX Titan X, Profiling, GPGPU, Symbiotic schedulers, Memory hierarchy contention

Índice general

CAPÍTULO 1

Introducción

En este capítulo se introduce el presente Trabajo Fin de Grado (TFG). En primer lugar, se presentan la arquitectura y el *software* que soportan la ejecución de las aplicaciones o *kernels* en las GPGPU de NVIDIA. Después, se introduce el soporte de estas GPGPU a la ejecución concurrente de kernels así como las limitaciones de este soporte. Entre otros aspectos, se introducen las tecnologías que permiten la compartición de los recursos de la GPGPU entre distintos kernels. Finalmente, se comenta la problemática sobre la contención en los recursos que la compartición entre kernels concurrentes puede causar y la necesidad de desarrollar planificadores que minimicen esta contención mientras maximizan la utilización de los recursos compartidos y las prestaciones de la GPU, lo que establece los objetivos principales de este TFG.

1.1 Unidades de procesamiento gráfico para cómputo de propósito general (GPGPU)

Las unidades de procesamiento gráfico (*Graphics Processing Units* o GPU) fueron ideadas inicialmente para el procesamiento gráfico de imágenes y video. Hoy en día existen distintas líneas de GPU especializadas o bien en aplicaciones de entretenimiento, de uso profesional o de computación científica [1]. Las GPU de cada una de estas líneas se implementan con distintas optimizaciones con la finalidad de obtener el mayor rendimiento posible en el ámbito correspondiente. En este TFG nos centraremos en el ámbito del uso de las GPU para computación científica.

Originalmente, las GPU fueron diseñadas con la finalidad de generar *frames*, es decir, matrices de píxeles, a gran velocidad para ser visualizados mediante una pantalla. Es por ello que su arquitectura es excelente para acelerar de un modo considerable la computación de operaciones sobre vectores y matrices. 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].

Debido a su excelencia en las mencionadas capacidades de cómputo, las GPU han tomado un rol importante en el ámbito de la computación de altas prestaciones (*High-Performance Computing* o HPC) como aceleradores de cómputo de propósito general (*General Purpose Graphics Processing Units* o GPGPU). En este sentido, las GPGPU se han demostrado muy exitosas en la ejecución de aplicaciones con un grado de paralelismo masivo [2]. Por ejemplo, un número considerable de las máquinas en la lista Top500 [3], la cual contiene una clasificación de los supercomputadores más potentes del mundo, implementan GPGPU.

1.2 Arquitectura de las GPGPU de NVIDIA

Las GPGPU actuales implementan varios multiprocesadores, cada uno de los cuales incluye un elevado (alrededor de 128) número de núcleos. Los núcleos se agrupan para ejecutar las mismas instrucciones de modo que siguen el modelo de procesamiento *Single Instruction Multiple Thread (SIMT)*. En las arquitecturas de las GPU NVIDIA, sobre las cuales se ha desarrollado el presente TFG, estos núcleos son conocidos como *CUDA cores* (o de ahora en adelante, simplemente cores).

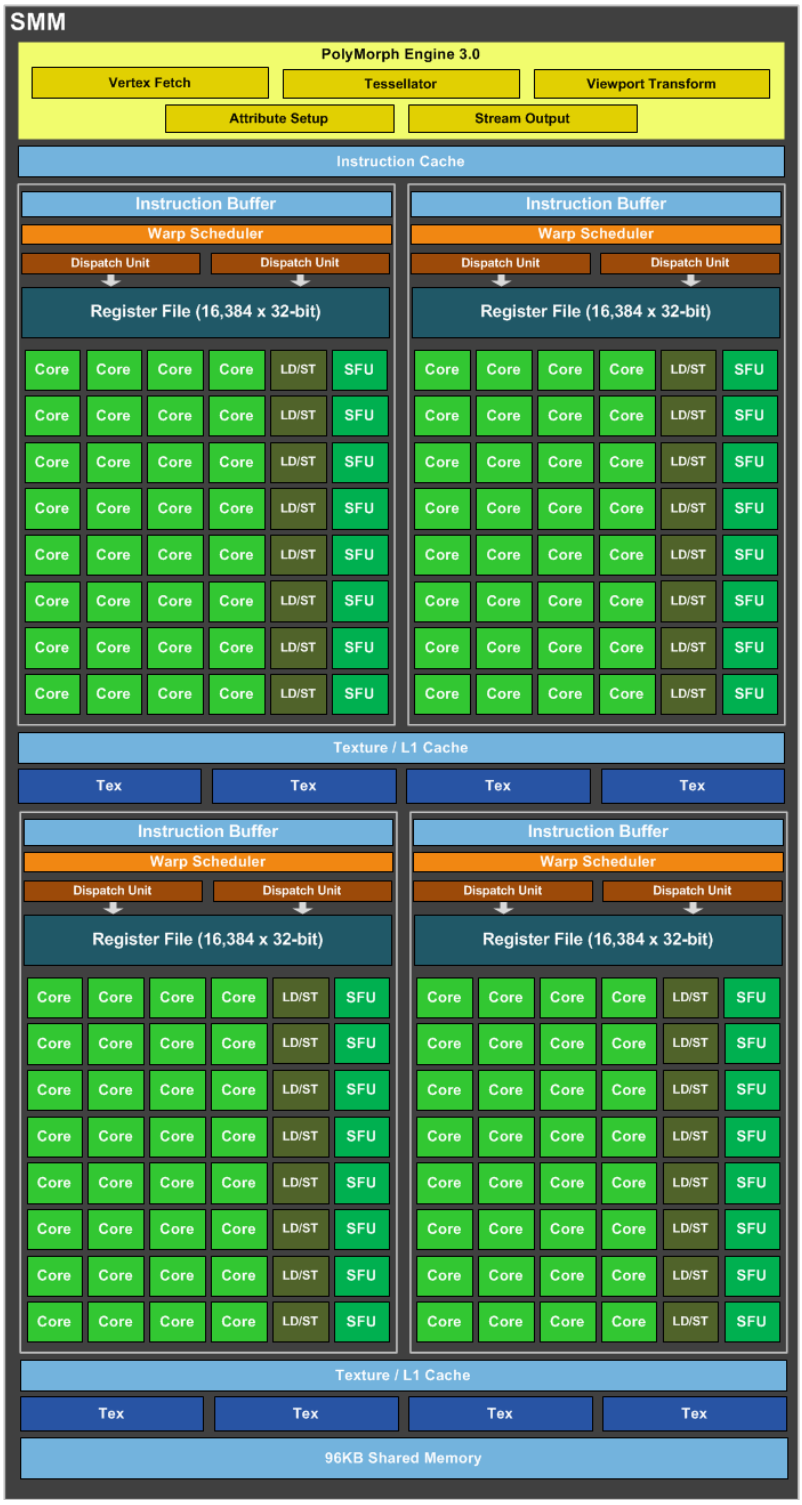

Figura 1.1: SM de arquitectura Maxwell de segunda generación. Fuente: NVIDIA Corporation.

Los multiprocesadores de una GPGPU, denominados *Streaming Multiprocessors* (SM) en los dispositivos de NVIDIA, incorporan recursos que se comparten entre los cores. Estos recursos son principalmente los bancos de registros, las unidades aritméticas (distintas en función del tipo de datos, por ejemplo, enteras o de coma flotante), las unidades de acceso a memoria, las memorias locales al multiprocesador y caches de instrucciones

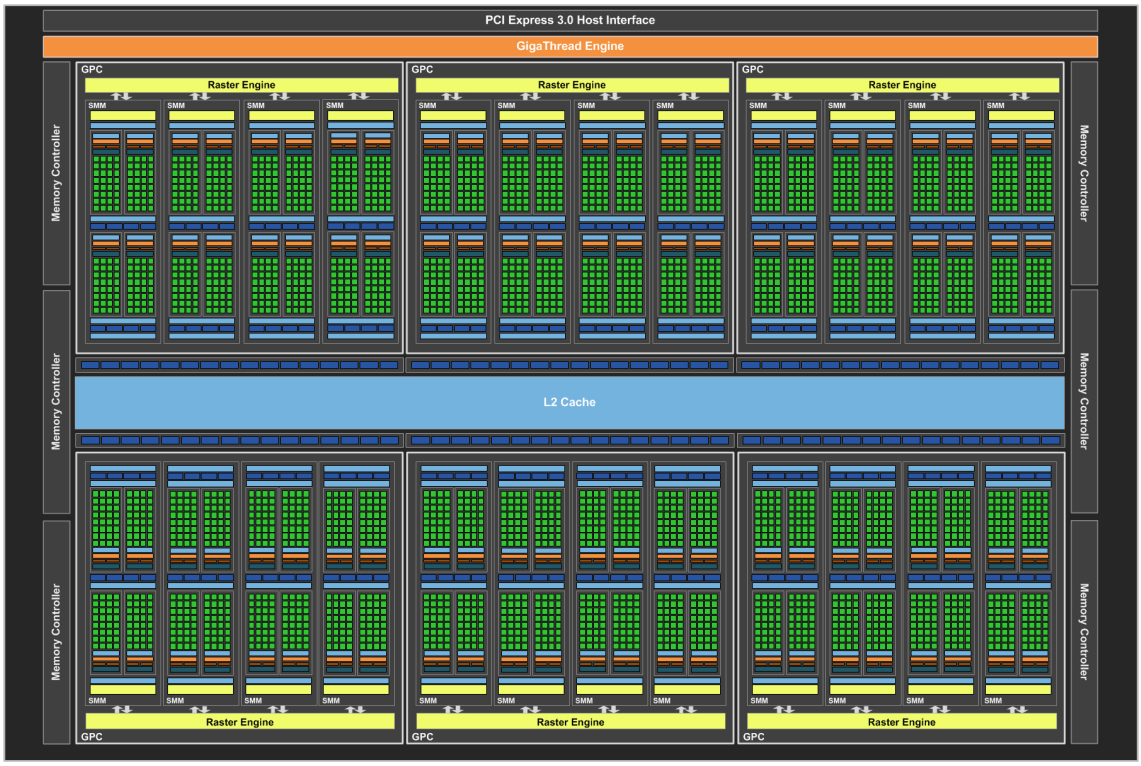

Figura 1.2: Tarjeta NVIDIA GeForce GTX Titan X. Fuente: NVIDIA Corporation.

y datos de primer nivel (L1). La figura 1.1 muestra el diagrama correspondiente a un SM de arquitectura Maxwell de segunda generación donde pueden apreciarse los cores y los recursos compartidos.

A su vez, los SM comparten entre ellos una memoria cache de segundo nivel (L2) común para datos e instrucciones, controladores de acceso a memoria principal y la memoria principal GDDR. Esto se puede apreciar en la figura 1.2, la cual muestra los SM disponibles en una tarjeta NVIDIA GeForce GTX Titan X y como estos comparten, entre otros recursos, la memoria cache de segundo nivel (L2).

1.3 Programación y ejecución de aplicaciones GPGPU con CUDA

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 GPGPU 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, al programador puede: i) reservar espacio en la memoria principal

de la GPGPU, ii) realizar transferencias de datos y código entre la memoria principal del computador y la de la GPGPU, y iii) programar el código o *kernel* que se ejecutará en la GPGPU.

Un kernel está formado por una multitud de *threads*. Estos threads 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 un core diferente [5].

1.4 Soporte a la ejecución concurrente en las GPGPU de NVIDIA

Con el lanzamiento de la arquitectura Fermi de NVIDIA, se incluyó el denominado *GigaThread Engine*, un planificador a dos niveles cuyo nivel externo asigna bloques a SM cuando hay recursos disponibles y en el nivel interno se encarga del lanzamiento de las instrucciones en cada core. Con este planificador, también se introdujo la capacidad de que un determinado proceso del sistema operativo pueda lanzar varios kernel en paralelo, permitiendo compartir entre varios kernel los recursos de una misma GPGPU, los cuales se desaprovecharían si la ejecución de los kernels fuera secuencial [6].

Posteriormente, la arquitectura Kepler incorporó la tecnología HyperQ, que introduce múltiples canales para que las transferencias de código y datos entre las memorias del procesador y la GPGPU puedan realizarse de forma paralela, evitando falsas dependencias [7]. Sin embargo, HyperQ sólo no permite que distintos procesos del sistema operativo puedan lanzar kernels en paralelo debido a limitaciones del sistema; por lo tanto, actualmente la ejecución concurrente de kernels es exclusiva a un solo proceso del sistema operativo. En otras palabras, kernels lanzados por distintos procesos no comparten nunca los recursos, lo que causa la serialización de kernels lanzados por distintos procesos [8] aun cuando realizan tareas independientes.

Para permitir que distintos procesos puedan lanzar kernels de manera concurrente e incluso si fuera necesario, colaborativa, como sería el caso de un programa de tipo MPI (*Message Passing Interface*), se puede utilizar el servicio MPS (*Multi Process Service MPS*). Este servicio consiste en una aplicación en segundo plano encargada de realizar la

técnica *Context Funneling*, la cual permite la mediación de un único proceso para lanzar concurrentemente en la GPGPU kernels de distintos procesos, evitando la mencionada serialización. El uso de MPS tiene varias limitaciones. La primera es que solo funciona en sistemas Linux. La segunda es que los kernels de distintos procesos podrían violar sus respectivos espacios de memoria. Finalmente, MPS no ofrece protección en caso de error, por lo que un error fatal en uno de los kernels provoca la terminación abrupta de todo el trabajo que se esté realizando en la GPGPU [9].

1.5 Objetivos del trabajo

Pese a las limitaciones mencionadas anteriormente, cabe esperar que el soporte para la ejecución concurrente de kernels avance aún más, lo que implicará la necesidad de desarrollar planificadores que elijan qué aplicaciones deben ejecutarse en la GPGPU en un momento dado.

El objetivo de este TFG es estudiar el comportamiento de los kernels cuando se ejecutan concurrentemente en una tarjeta NVIDIA GeForce GTX Titan X como primer paso en el diseño e implementación de un planificador que maximice las prestaciones de una carga de trabajo compuesta por varios kernels. Para ello, se caracterizará el comportamiento de los kernels tanto individualmente como en ejecución, con la intención de comprobar el impacto en las prestaciones de los recursos compartidos y las interferencias entre kernels accediendo a ellos.

Para alcanzar el objetivo principal de este TFG se han seguido los siguientes objetivos específicos:

1. Caracterizar el comportamiento de las aplicaciones desde el punto vista del consumo de recursos y de las prestaciones en términos de instrucciones por ciclo (IPC).

2. Analizar y cuantificar el impacto de la interferencia entre aplicaciones en las prestaciones individuales de los kernels.

3. Establecer una base para el diseño de futuros algoritmos de planificación que permitan maximizar la utilización de los recursos de las GPGPU e incrementar las prestaciones.

CAPÍTULO 2

Plataforma Experimental

En este capítulo se presentan las características de la GPGPU utilizada para la experimentación realizada en este trabajo, así como el software del *host*, los contadores de prestaciones y las herramientas que se utilizan para extraer los datos de los mismos.

2.1 *NVIDIA GeForce GTX Titan X*

2.1.1. Arquitectura Maxwell de segunda generación

Como se ha comentado anteriormente, en el desarrollo de este TFG se ha hecho uso de una tarjeta NVIDIA GeForce GTX Titan X. Esta GPGPU está implementada con la segunda generación de la arquitectura Maxwell de NVIDIA.

Un SM de la arquitectura Maxwell está dividido en 4 unidades 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 2.1 muestra un resumen de las capacidades de la arquitectura Maxwell de segunda generación [10].

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 2.1: Capacidades de la arquitectura Maxwell de segunda generación.

La jerarquía de memoria 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 texturas, 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 [11].

2.1.2. Especificaciones técnicas

La NVIDIA GeForce GTX Titan X es una GPGPU de la gama de productos GeForce, cuyo principal público es el consumidor de videojuegos. Sin embargo, el grado de compatibilidad de esta GPGPU con CUDA, el número de SM que ofrece, y su capacidad de memoria principal la hacen un sistema experimental adecuado para este TFG, pues su gran cantidad de recursos facilita el desarrollo de experimentos donde varios kernels los

comparten en paralelo. La tabla 2.2 presenta las especificaciones técnicas de esta tarjeta [12].

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 2.2: Especificaciones técnicas de la tarjeta NVIDIA GeForce GTX Titan X.

2.2 *Host, Software y drivers*

Para la experimentación se ha instalado la tarjeta en un servidor con sistema operativo *16.04.1-Ubuntu kernel 4.10.0-14-generic* y una instalación de *CUDA SDK V8.0.61*, la versión del *driver* de NVIDIA utilizada es la 375.66.

Incluidas en CUDA SDK se encuentran las herramientas de *profiling* (evaluación de prestaciones) *nvprof*, así como el entorno de desarrollo *Nsight* y el *NVIDIA Visual Profiler (nvvp)*. Todas las herramientas mencionadas son utilizadas para la recogida de datos de los contadores de prestaciones *hardware* de la GPGPU, listados a continuación.

2.3 Contadores hardware

2.3.1. *elapsed_cycles_sm*

Este contador acumula la suma del tiempo de ejecución de cada SM en ciclos [13]. Por lo tanto, para obtener el tiempo global de ejecución en ciclos (*execution_cycles*) se divide el valor de este contador por el número de SM activos (i.e. que han participado en la ejecución del kernel). En consecuencia, para calcular el tiempo de ejecución del kernel en segundos se utiliza la ecuación

$$execution\_time = \frac{elapsed\_cycle\_sm}{GPUClock \times \#SM}, \tag{2.1}$$

donde $GPUClock$ representa la frecuencia de reloj de la GPGPU.

2.3.2. *inst_executed*

El resultado de este contador nos da una media de las instrucciones totales ejecutadas en cada SM activo [14].

2.3.3. Accesos y fallos en L2

La cache de L2 está particionada en dos sectores, donde cada sector dispone de un contador para lecturas (*l2_subp**[0|1]**_total_read_sector_queries*) y otro para escrituras (*l2_subp**[0|1]**_total_write_sector_queries*). La suma de estos dos contadores da el número total de accesos a la cache de L2.

Además, existen otros dos contadores para cada sector que miden el número de fallos de L2 (*l2_subp**[0|1]**_[write|read]_sector_misses*). En otras palabras, estos contadores permiten medir el número de accesos a la memoria principal DRAM [15].

Accesos a L1

Al igual que en la cache de L2, las caches de datos de L1 se particionan en dos sectores con un contador de peticiones para cada uno (*tex**[0|1]**_cache_sector_queries*). Al compartir la cache de datos de L1 su hardware con la cache de texturas, el número de accesos y fallos en esta cache también se ve influido por el uso de los threads del subsistema de texturas [15].

Distribución del tiempo de ejecución

Las herramientas de evaluación de prestaciones proporcionadas por NVIDIA permiten configurar los contadores hardware para tomar muestras del estado de los warp activos en la GPGPU. De este modo, es posible saber qué porcentaje del tiempo ha pasado cada kernel lanzando instrucciones y las causas que han impedido el lanzamiento de instrucciones en cada ciclo [16].

CAPÍTULO 3

Metodología

En este capítulo se presentan las aplicaciones escogidas para llevar a cabo este trabajo. Además, se comenta el procedimiento seguido para el lanzamiento de las aplicaciones y el acceso a los contadores de prestaciones. Finalmente se explican los índices de prestaciones que se evaluarán en capítulos posteriores.

3.1 Aplicaciones y cargas utilizadas

En esta sección se describen las aplicaciones que se utilizarán en el estudio de caracterización asi como sus parámetros y tamaños de carga. Las aplicaciones pertenecen a las suites *lonestarGPU2.0* [17], *SHOC* [18] y *Rodinia* [19, 20].

La tabla 3.1 muestra un resumen de las características de las aplicaciones elegidas. La tabla muestra en cada columna, de izquierda a derecha, para cada aplicación, el nombre del kernel que lanza, el tamaño de grid en número de bloques, el tamaño de bloque en número de threads, el número de lanzamientos del kernel que realiza la aplicación, el número de instrucciones que ejecuta cada kernel por SM, y el tiempo medio de ejecución de todas las instancias del kernel lanzadas por la aplicación. En el caso de la aplicación QTC el tamaño de grid y el tiempo de ejecución de los kernels varia notablemente, por lo que se ha optado por especificar el rango de variación de estas dos características en la tabla.

Aplicación	**kernel**	**Grid size**	**Block size**	**#Lanzamientos**	**#instr. (M)**	T_{ex}**(ms)**
BH	ForceCalculationKernel	120	256	360	197	10.81
QTC	QTC_Device	37–256	64	446	285	0.02–23.41
HW	kernel	51	256	103	279	11.92
LAVA	kernel_gpu_cuda	125	128	150	43	11.86
SRAD	reduce	450	512	10000	1	0.02
LEUKO	IMGVF_kernel	36	320	100	345	16.17

Tabla 3.1: Características de las aplicaciones estudiadas.

3.1.1. *Barnes-Hutt Simulation (BH)*

Esta aplicación pertenece a la suite lonestarGPU2.0. Es una implementación CUDA del algoritmo de Barnes et al. para el cálculo de la fuerza ejercida por la gravedad en una agrupación de cuerpos [21]. La carga utilizada es de dieciséis mil cuerpos y ciento veinte pasos o iteraciones.

3.1.2. *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 [22]. La carga utilizada es la

correspondiente al set de datos predeterminado que se genera al lanzar la aplicación con el parámetro "*–size 2*".

3.1.3. *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 video de imagen por ultrasonido [23]. Para ejecutar la aplicación se usa como carga un video de ciento cuatro fotogramas proporcionado por la suite y se indica por parámetro que se analice la totalidad del mismo.

3.1.4. *LavaMD (LAVA)*

Esta aplicación mide las fuerzas ejercidas por una agrupación de partículas en un espacio amplio [24]. Para lanzarla se utiliza el parametro "*-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 150 veces.

3.1.5. *Speckle Reducing Anisotropic Diffusion (SRAD)*

Esta aplicación se utiliza para la reducción de ruido en imágenes de ultrasonido [25]. La aplicación se lanza con los siguientes parametros "*10000 0.5 502 458*".

3.1.6. *Leukocyte Tracking (LEUKO)*

Esta aplicación cuenta el número de leucocitos en tiempo real [26]. La carga utilizada es un video de 100 fotogramas proporcionado por la suite, indicándose por parámetro que se analice la totalidad del mismo.

LEUKO sufre un error fatal si se ejecuta concurrentemente con LAVA, por lo cual no se han podido recoger datos sobre sus interferencias.

3.2 Lanzamiento de las aplicaciones

Las aplicaciones se lanzan tanto en los experimentos en ejecución individual como en los experimentos en ejecución concurrente mediante un *script* del *shell bash*. Antes

de lanzarse las aplicaciones el script se asegura de iniciar el servicio MPS (*Multi Process Service*), el cual permite que los kernels de diversas aplicaciones compartan los recursos de la GPGPU.

3.3 Acceso a los contadores hardware

El acceso a los contadores hardware se realiza mediante el uso de la herramienta nvprof. Al terminar la ejecución esta herramienta genera un fichero de resultados que puede abrirse con la herramienta nvvp con la finalidad de visualizar gráficamente los resultados y exportarlos a una hoja de cálculo para ser analizados posteriormente.

Aunque en teoría es posible utilizar la herramienta nvprof tanto en ejecución individual como en ejecución concurrente, en la práctica el sistema no permite contar separadamente los eventos causados por kernels distintos. Esta limitación hace que el único índice de prestaciones que se puede obtener durante la ejecución concurrente sea el IPC.

3.4 Índices de prestaciones usados en la caracterización

3.4.1. Instrucciones por Ciclo (IPC)

Para obtener el IPC de cada kernel se usa la ecuación

$$IPC = \frac{inst\_executed}{execution\_cycles}, \tag{3.1}$$

donde *inst_executed* y el tiempo de ejecución *execution_cycles* se obtienen tal como se explica en la sección 2.3.

3.4.2. Tasa de accesos a la cache de L1 de datos

El número total de accesos a la cache de L1 de datos se calcula mediante la suma

$$tex\_queries = hardwaretex0\_cache\_sector\_queries + tex1\_cache\_sector\_queries, \tag{3.2}$$

donde los contadores $tex0\_cache\_sector\_queries$ y $tex1\_cache\_sector\_queries$ representan los accesos correspondientes a los dos sectores de esta cache.

Puesto que una aplicación lanza n kernels a lo largo de su ejecución, para obtener la tasa de accesos global es necesario acumular el total de accesos generados por los kernels de la aplicación y dividir este valor por el tiempo de ejecución acumulado por

estos kernels. Así,

$$TA_{L1D} = \frac{\sum_{i=1}^{n} tex\_queries_i}{\sum_{i=1}^{n} execution\_time_i}. \tag{3.3}$$

Este índice está relacionado con la interferencia que pueden sufrir los warp de distintos bloques que se encuentran en el mismo SM.

3.4.3. Tasa de accesos a la cache de L2

La tasa de accesos a la cache de L2 se obtiene a partir del número total de peticiones que se realizan a esta caché sumando los contadores *l2_subp0_total_read_sector_queries*, *l2_subp1_total_read_sector_queries*, *l2_subp0_total_write_sector_queries* y *l2_subp1_total_write_sector_queries* de manera análoga a la tasa de accesos a L1. Es decir,

$$TA_{L2} = \frac{\sum_{i=1}^{n} l2\_queries_i}{\sum_{i=1}^{n} execution\_time_i}. \tag{3.4}$$

Con los valores obtenidos se puede estimar cómo se interfieren los kernels ejecutándose en distintos SM.

3.4.4. Tasa de accesos a la memoria principal

Los accesos a la DRAM se realizan cuando se produce un fallo en la cache de L2. Para medirlos, se utilizan los contadores *l2_subp0_read_sector_misses*, *l2_subp1_read_sector_misses*, *l2_subp0_write_sector_misses* y *l2_subp1_write_sector_misses*. De esta manera,

$$TA_{MM} = \frac{\sum_{i=1}^{n} l2\_misses_i}{\sum_{i=1}^{n} execution\_time_i}. \tag{3.5}$$

3.4.5. Tasa de fallos de L1

La tasa de fallos en la cache de L1 de datos se puede calcular dividiendo el número total de accesos a L2 por el número total de accesos a L1. Así,

$$TF_{L1} = \frac{\sum_{i=1}^{n} l2\_queries_i}{\sum_{i=1}^{n} tex\_queries_i}. \tag{3.6}$$

3.4.6. Tasa de fallos de L2

El porcentaje de los accesos que fallan en L2 se obtiene dividiendo los fallos totales en L2 entre el total de accesos a este nivel de la cache. De este modo,

$$TF_{L2} = \frac{\sum_{i=1}^{n} l2\_misses_i}{\sum_{i=1}^{n} l2\_queries_i}. \tag{3.7}$$

3.4.7. Distribución del tiempo de ejecución

El entorno de desarrollo Nsight, enfocado a la optimización del código, proporciona herramientas que permiten desensamblarlo y automatizar el análisis del comportamiento de los kernels en ejecución. Estas herramientas permiten discernir los posibles motivos por los cuales una instrucción no es capaz de ejecutarse en el SM (causando un *stall*) y generar un informe con la distribución de cada tipo de stall en el tiempo de ejecución. Analizando estas distribuciones se pueden observar las causas más comunes por las cuales los warps no pueden lanzar instrucciones.

CAPÍTULO 4

Resultados Experimentales

En este capítulo se presentan los principales resultados experimentales obtenidos en el desarrollo de este TFG. En primer lugar, se realizaron una serie de tests para comprobar la concurrencia en la GPGPU de varios kernels lanzados en paralelo. En segundo lugar se caracterizan el comportamiento individual de las aplicaciones con el objetivo de detectar los posibles puntos de contención. Finalmente se analizan los resultados de las aplicaciones en ejecución concurrente, mostrando la necesidad de planificadores que tengan en cuenta el comportamiento de las aplicaciones y la contención para maximizar las prestaciones.

4.1 Tests preliminares de paralelismo

Con la finalidad de comprobar si las aplicaciones candidatas para este trabajo son adecuadas para el mismo, es decir, si se ejecutan en paralelo, o por el contrario, ocupan tantos recursos que se serializan al ejecutarse en la GPU se realizaron diferentes tests, comentados en esta sección.

4.1.1. Test de paralelismo por parejas

En primer lugar, para cada aplicación i, se midieron el tiempo de ejecución individual T_i y el tiempo de ejecución concurrente con cada una de las parejas posibles (incluido consigo mismas). De este modo, el tiempo de ejecución concurrente de la aplicación i con la aplicación j, siendo i y j dos aplicaciones cualesquiera, se define como $T_{i,j}$. Los resultados de estas mediciones se usaron para calcular el *Individual Speedup* (IS) de una aplicación i cuando se ejecuta con otra aplicación j mediante la ecuación

$$IS_{i,j} = \frac{T_{i,j}}{T_i}. \tag{4.1}$$

Nótese que $IS_{i,j}$ es siempre mayor que 1 ya que la ejecución concurrente de i con j provoca interferencias que alargan el tiempo de ejecución $T_{i,j}$. Por otro lado, para que exista algún paralelismo entre i y j debe cumplirse la condición

$$IS_{i,j} < \frac{T_i + T_j}{T_i}. \tag{4.2}$$

La tabla 4.1 muestra el $IS_{i,j}$ para cada aplicación en la fila i cuando se ejecuta con otra aplicación en la columna j En el caso de la ejecución concurrente de LAVA y LEUKO, no fue posible obtener resultados debido al error mencionado en la sección 3.1. De los resultados que sí pudieron obtenerse, se comprobó que todas las parejas, excepto BH al ejecutarse con QTC, HW, LAVA y SRAD cumplían la condición de la ecuación 4.2. Puesto

Aplicación	BH	QTC	HW	LAVA	SRAD	LEUKO
BH	1.11	1.68	1.82	2.33	1.84	1.08
QTC	1.05	1.22	1.22	1.88	1.26	1.18
HW	1	1.14	1.77	1.76	1.44	1.02
LAVA	1.21	1.13	1.5	1.85	1.41	–
SRAD	1.16	1.29	1.24	1.93	1.63	1.18
LEUKO	1.52	1.48	1.27	–	1.28	1.44

Tabla 4.1: Resultados obtenidos en el test de paralelismo por parejas.

que el presente test no resulta infalible, ya que pueden existir sobrecargas que eviten que se cumpla la mencionada condición, se realizaron tests adicionales, comentados más abajo.

4.1.2. Test de límite del paralelismo

Además de los tests por parejas, se realizaron tests donde un número m de instancias de una determinada aplicación i se ejecuta en paralelo, obteniéndose el tiempo de ejecución concurrente de una aplicación i consigo misma para m instancias $T_i(m)$. Con este tiempo y el tiempo de ejecución individual de la aplicación T_i se calcula la aceleración respecto a la ejecución en secuencial de las m instancias, es decir

$$S_i(m) = \frac{T_i \times m}{T_i(m)}. \tag{4.3}$$

Es fácil comprobar que $S_i(m)$ crece, aunque no linealmente debido a las interferencias, con m siempre que las m instancias de la aplicación i puedan ejecutarse en paralelo, llegando a un valor máximo $S_i(max)$ cuando la GPGPU no tiene suficientes recursos para ofrecer más paralelismo efectivo. La tabla 4.2 muestra ese valor máximo para cada una de las aplicaciones estudiadas, así como el número de instancias requeridas para alcanzar ese valor. Se observa que la mitad de las aplicaciones alcanzan la máxima aceleración con alrededor 4 instancias, mientras que el resto la alcanzan con un valor mucho mayor, lo que demuestra la gran capacidad de las GPGPU actuales para ejecutar aplicaciones concurrentemente.

Aplicación	#instancias	S_i(max)
BH	5	2.54
QTC	11	2.57
HW	4	1.52
LAVA	16	2.51
SRAD	8	1.75
LEUKO	4	1.74

Tabla 4.2: Resultados obtenidos en el test de nivel de paralelismo.

4.1.3. Test visual de paralelismo

También se procedió a una inspección visual de la ejecución con la herramienta nvvp para comprobar el solapamiento durante la ejecución de kernels de aplicaciones en ejecución concurrente. La figura 4.1 muestra una ventana de la herramienta nvvp donde en

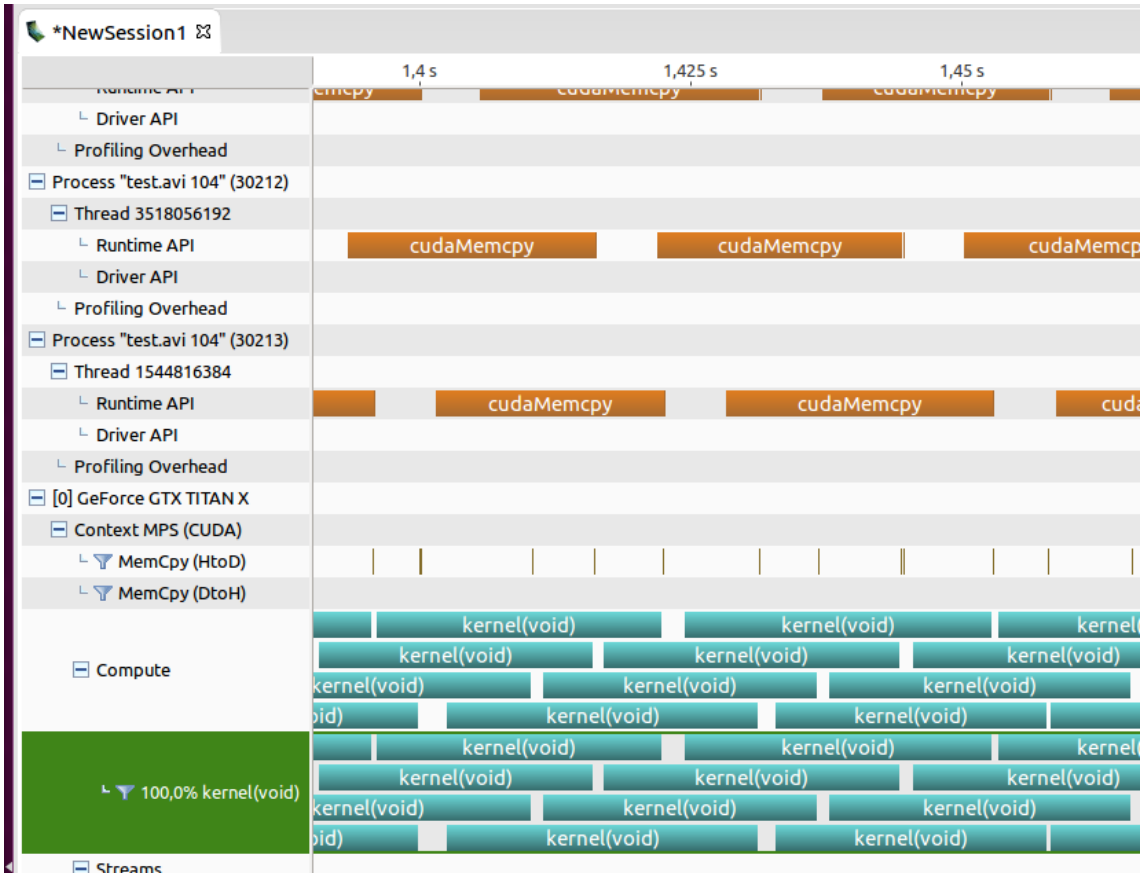

Figura 4.1: Ventana de nvvp mostrando la ejecución simultanea de cuatro instancias de HW.

el campo *Compute* se observa que en todos los instantes del tiempo de ejecución la tarjeta está ejecutando al menos 3 kernels de la aplicación HW.

4.2 Caracterización de las aplicaciones en ejecución individual

4.2.1. IPC

La figura 4.2 muestra el IPC medio de los kernels lanzados por las diferentes aplicaciones en ejecución individual. En general los IPC de las aplicaciones estudiadas se encuentran en valores alrededor de la unidad, excepto el IPC de LAVA, con un valor alrededor de 0,2, y el de SRAD, cuyo IPC es prácticamente igual a 1,75. Como se verá en secciones posteriores, el bajo IPC de LAVA se debe a contención en el acceso a las unidades funcionales de coma flotante de doble precisión, cuyas prestaciones son muy bajas en las tarjetas orientadas a *gaming* como es el caso de la NVIDIA Titan X. Con respecto a SRAD, hemos comprobado que sus excepcionales prestaciones se deben a que es la aplicación que lanza un mayor número de warps por kernel, lo que le permite ocultar mejor las latencias de acceso a memoria y mejorar por tanto su IPC [27].

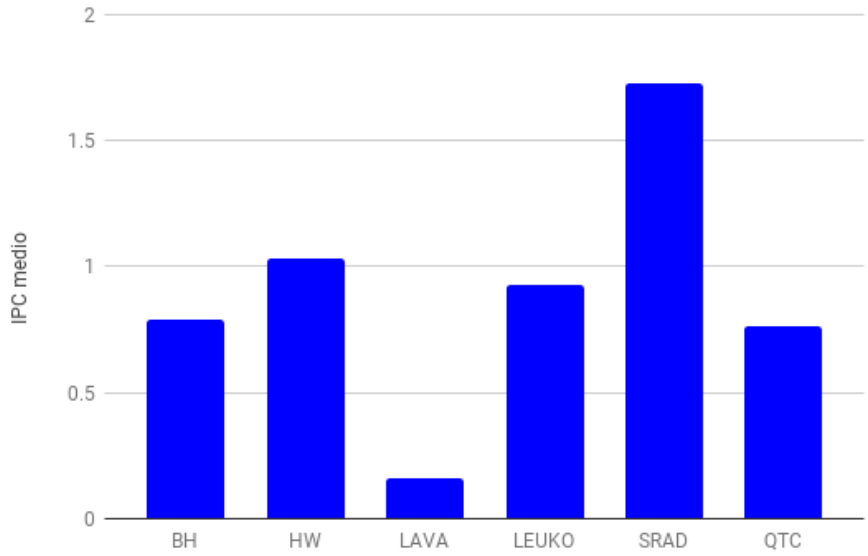

Figura 4.2: IPC medio de los kernels de las aplicaciones.

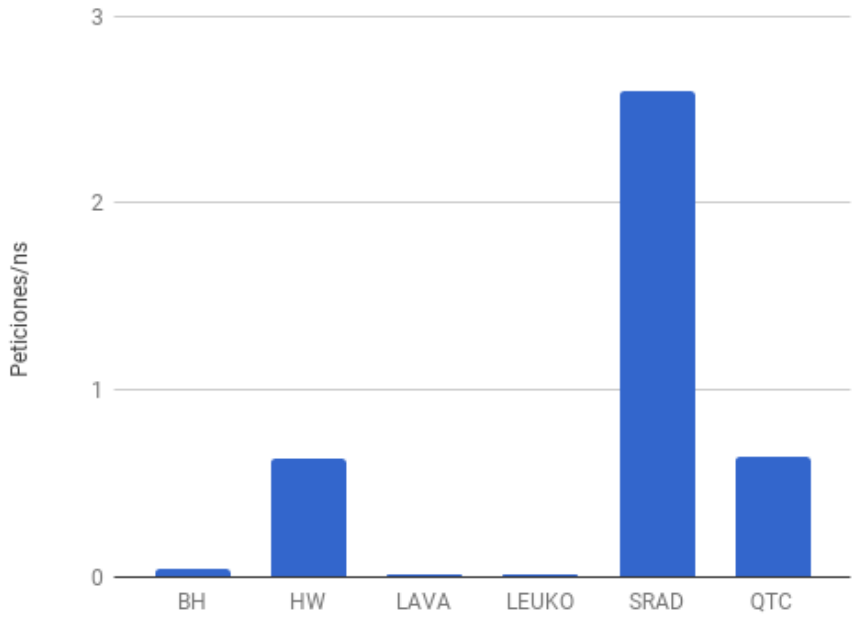

Figura 4.3: Tasa de accesos a memoria principal de los kernels de las aplicaciones.

4.2.2. Tasas de accesos y fallos

La figura 4.3 presenta las tasas de acceso medias (en peticiones/ns) a memoria principal debidas a los kernels lanzados por las diferentes aplicaciones. Como se puede observar, SRAD es la aplicación que mayor contención causa en la memoria principal, mientras que HW y QTC son aplicaciones con una tasa de accesos intermedia a esta estructura de memoria. Finalmente, BH, LAVA y LEUKO hacen un uso escaso de la memoria principal (por debajo de 0,5 peticiones/ns).

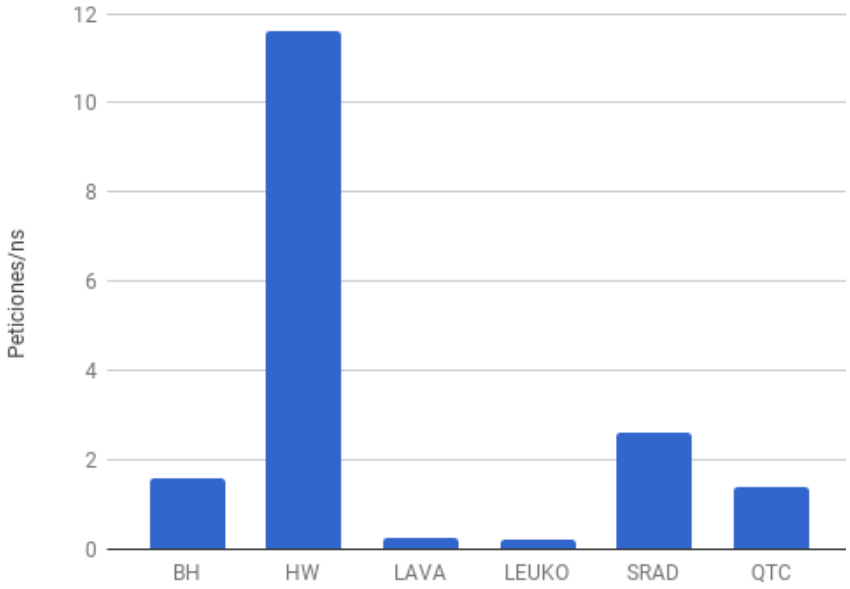

Figura 4.4: Tasa de accesos a L2.

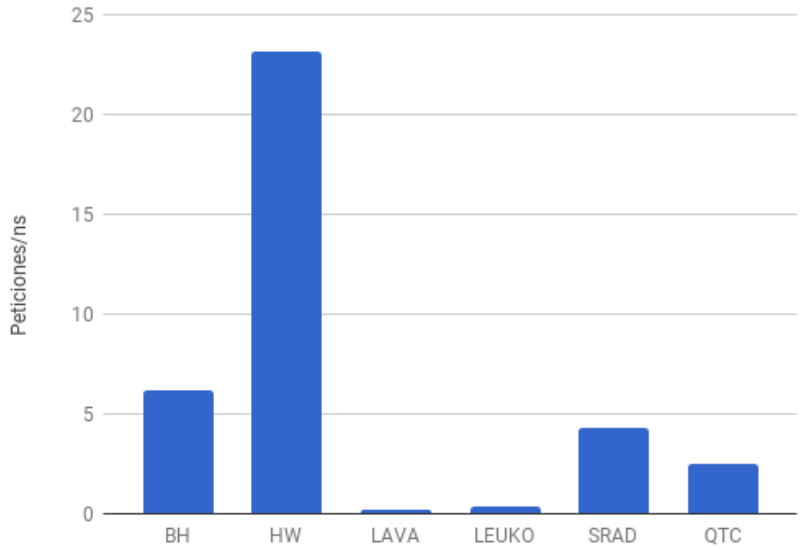

Figura 4.5: Tasa de accesos a la cache de L1 de datos.

En cuanto a la tasa de acceso a la cache de L2 (figura 4.4), HW es la aplicación que ejerce mayor presión en esta cache. En comparación, las tasas de acceso de BH, SRAD y QTC se pueden clasificar como intermedias, mientras que LAVA y LEUKO presentan una baja tasa de acceso a L2 (alrededor de 0,2 peticiones/ns).

Finalmente, en el acceso a la cache de L1 de datos (figura 4.5), todos los kernels presentan un comportamiento relativo parecido al mostrado en L2. Por ejemplo, HW sigue siendo la aplicación con mayor tasa de acceso con diferencia, mientras LAVA y LEUKO apenas consumen ancho de banda en L1. Por otro lado, el orden de magnitud de las tasas de acceso a L1 y L2 es similar. Esto demuestra que existe un porcentaje significativo de ac-

cesos que fallan en L1 (ver tabla 4.3), lo que en general es debido a la menor localidad de los datos en los kernels GPGPU en comparación a las aplicaciones CPU convencionales.

Aplicación	L2 miss rate	L1 miss rate
BH	2.70 %	25.42 %
HW	5.44 %	50.04 %
LAVA	4.20 %	99.96 %
LEUKO	5.09 %	52.32 %
SRAD	100 %	60.18 %
QTC	46.58 %	55.12 %

Tabla 4.3: Tasas de fallo.

La tabla 4.3 también muestra que LAVA tiene una tasa de fallos en L1 prácticamente igual al 100 %. Cabe preguntarse porqué una aplicación con tan alta tasa de fallos en L1 ejerce tan poca presión sobre niveles inferiores de la jerarquía de memoria. Esta situación también ocurre, aunque en menor grado, con LEUKO. La razón en ambos casos se encuentra, como se verá más abajo, en el uso intensivo que hacen estas dos aplicaciones de operaciones de coma flotante de doble precisión, cuyas prestaciones son muy bajas en nuestra plataforma experimental. Esto conlleva una ralentización de la ejecución de estas aplicaciones que a su vez reduce sus tasas de acceso efectivas.

4.2.3. Distribución del tiempo de ejecución

La figura 4.6 presenta la distribución del tiempo de ejecución separando los ciclos dedicados a lanzar instrucciones (en verde en la parte inferior) de las diferentes causas de stall (resto de categorías). Una de las causas más importantes es la denominada *Execution dependency*, la cual ocurre cuando una instrucción no puede lanzarse debido a que al menos uno de sus operandos aún no ha sido producido por una operación anterior. La frecuencia de esta causa es especialmente significativa en LAVA y LEUKO. Para averiguar la razón, mediante el uso de la herramienta Nsight, comprobamos que ambas aplicaciones ejecutan una gran cantidad de operaciones de coma flotante de doble precisión. Por otro lado, la tarjeta NVIDIA Titan X usada en nuestros experimentos sólo implementa 4 unidades aritméticas para este tipo de datos por SM, lo que limita las prestaciones de las operaciones de coma flotante de doble precisión por dos razones: i) evita que las unidades SIMT puedan emitir dos instrucciones por ciclo, y ii) hace que las operaciones de coma flotante de doble precisión sean 32 veces más lentas que las operaciones de enteros y coma flotante de simple precisión [28]. Por todas estas razones, las prestaciones de

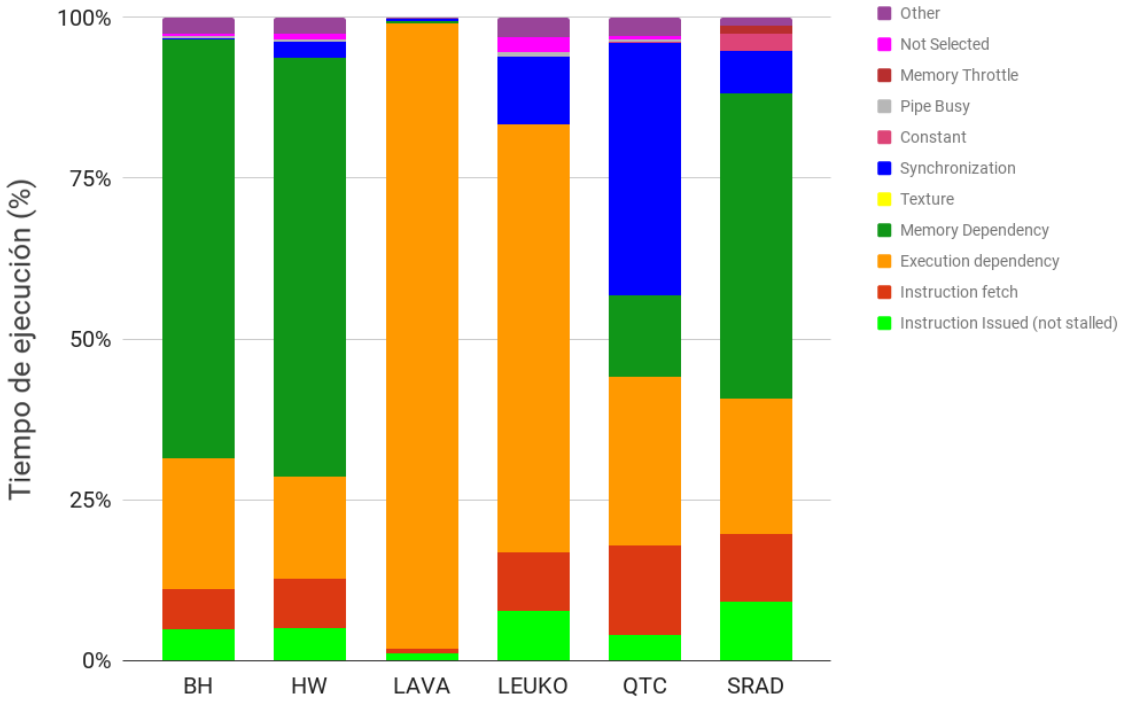

Figura 4.6: Distribución del tiempo de ejecución.

LAVA y LEUKO se ven especialmente perjudicadas, lo que a su vez reduce sus tasas de acceso a la jerarquía de memoria, como se comprueba en los resultados presentados en secciones anteriores.

Las restantes aplicaciones no hacen un uso tan intensivo de la coma flotante de doble precisión y por tanto no sufren la mencionada pérdida de prestaciones. La figura muestra que en esos casos, la principal causa de stall es generalmente *Memory dependency*. Este stall se produce cuando una instrucción no se puede lanzar porque uno de los operandos no está disponible y está siendo servido por la jerarquía de memoria. Este hecho demuestra que la jerarquía de memoria es un recurso crítico en las GPGPU y por lo tanto una buena política de planificación debería tener en cuenta la contención de múltiples kernels en el acceso a memoria con el objetivo de reducir el número de stalls y mejorar las prestaciones.

Por último, algunas cargas presentan esperas significativas debido a sincronización (p.e. QTC) o a la búsqueda de instrucciones. Nótese que la reducción del número de stalls causados por ambas razones se encuentra fuera del alcance del tipo de planificador al que se enfoca este trabajo.

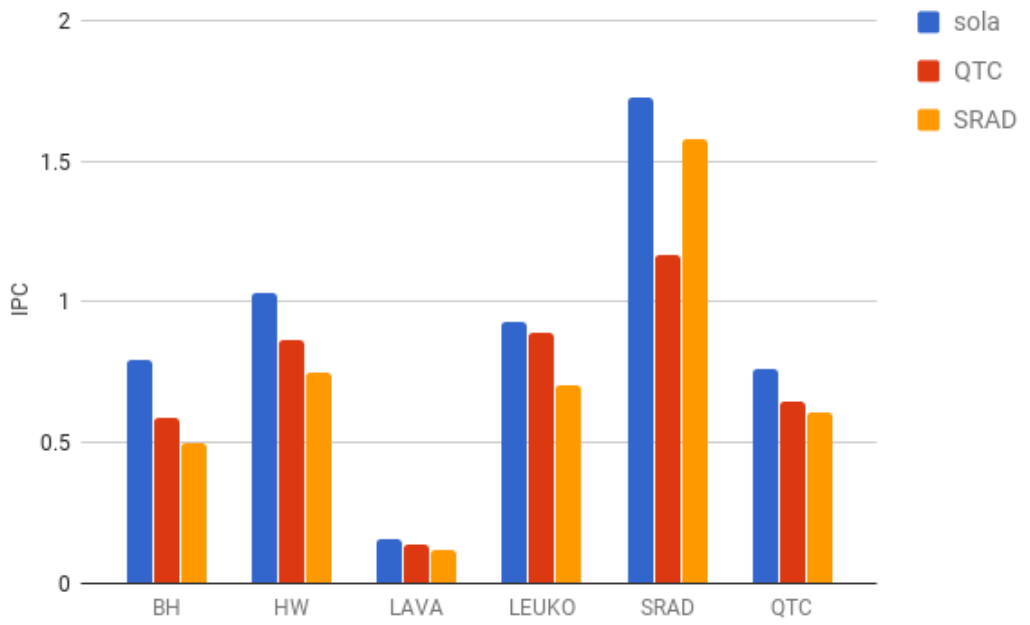

Figura 4.7: IPC en ejecución concurrente variando la contención en memoria principal.

4.3 Estudio de interferencias en ejecución concurrente

Una vez caracterizado el comportamiento de las aplicaciones en ejecución individual, esta sección caracteriza el impacto en las prestaciones que sufre cada aplicación cuando se ejecuta en paralelo dentro de la GPGPU. El objetivo es comprobar como diversos niveles de contención en los distintos puntos de la jerarquía de memoria degradan las prestaciones. Para realizar este estudio, se han elegido, para cada punto de contención entre SMs (es decir, para L2 y memoria principal), dos aplicaciones con un nivel intermedio y alto de consumo de ancho de banda en el correspondiente punto de contención. Esta selección se ha realizado en base a los resultados del estudio de caracterización en ejecución individual presentados en la sección 4.2.2.

Para el estudio de la contención en memoria principal se ha seleccionado la aplicación SRAD, la cual presenta una alta tasa de accesos a memoria principal, y la aplicación QTC con una tasa de accesos considerada intermedia. Los resultados (ver figura 4.7) demuestran que un mayor nivel de contención produce una caída de prestaciones generalizada para todas las aplicaciones. Sin embargo, es necesario tener en cuenta que aplicaciones que presentan bajas prestaciones incluso cuando se ejecutan en solitario, como LAVA, pueden ser candidatas idóneas para ejecutar concurrentemente con aplicaciones que ejerzan una alta contención en memoria principal, ya que en términos absolutos la pérdida de prestaciones de LAVA es mucho menor que la que pueda sufrir BH, por ejemplo.

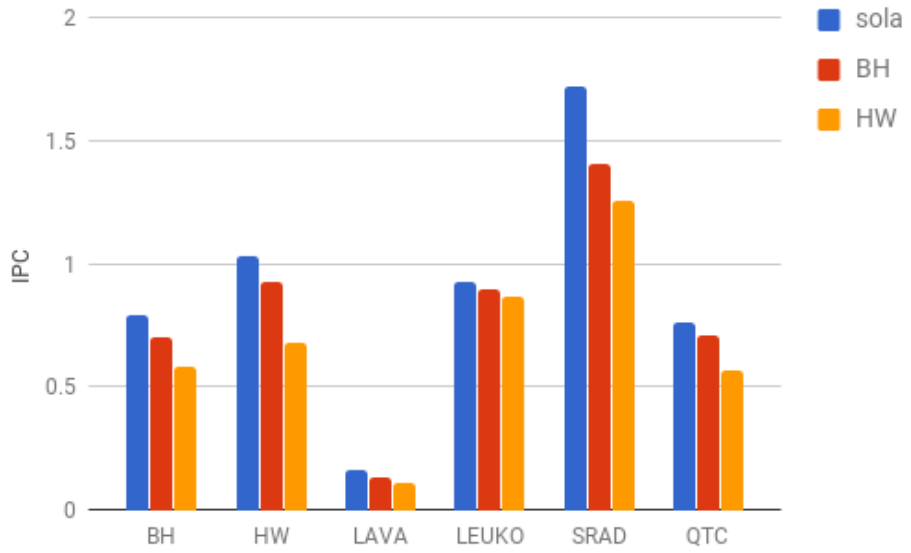

Figura 4.8: IPC en ejecución concurrente variando la contención en L2.

La única excepción a la mencionada tendencia generalizada la protagoniza SRAD cuando se ejecuta consigo misma. Hemos comprobado que el uso de los recursos que hace SRAD en los SMs permite lanzar a los dos instancias cuando se ejecutan en paralelo hasta un total de 900 bloques (450×2, ver 3.1) de 16 warps a la GPGPU, lo que tiene un impacto positivo en la ocultación de la latencia de los accesos a memoria.

Respecto al impacto de la contención en L2, se han elegido las aplicaciones BH y HW, con unas tasas de acceso a memoria intermedia y alta, respectivamente. Los resultados de la figura 4.8 indican que la cache de L2, aún siendo una estructura de memoria mucha más rápida que la memoria principal, puede presentar niveles de contención con un impacto en las prestaciones similar al de la contención en DRAM. Sin embargo, en el caso de L2, el nivel de impacto varía entre aplicaciones. Así, LEUKO apenas sufre a medida que se aumenta la contención en L2, mientras que HW puede reducir sus prestaciones a prácticamente la mitad. Todos estos aspectos deben ser tenidos en cuenta por un planificador para maximizar la productividad en la ejecución de varias aplicaciones.

Para completar el estudio, la figura 4.9 muestra, para cada aplicación i, la media geométrica de los ratios $T_i/T_{i,j}$, donde j es cualquier otra aplicación, es decir,

$$IPC\_degradation_i = \sqrt[n]{\prod_{j=1}^{n} \frac{T_i}{T_{i,j}}}. \tag{4.4}$$

Este valor representa, para una aplicación i, el nivel de interferencia que esta aplicación ejerce de media sobre otras aplicaciones en ejecución concurrente. Como se puede apreciar, la variación en la degradación de prestaciones que cada aplicación puede causar es significativa, siendo BH la aplicación que genera menos interferencia y LAVA la que más

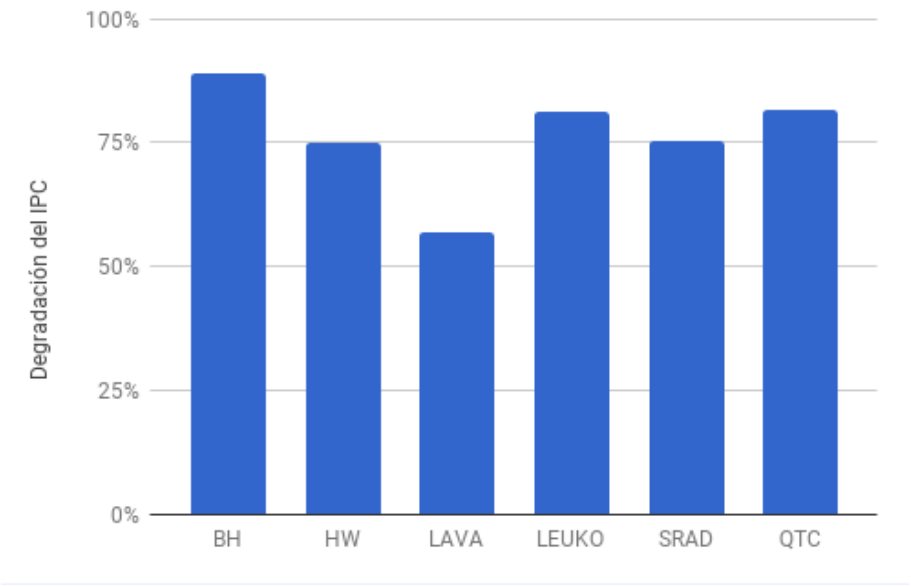

Figura 4.9: Media geométrica de la degradación del IPC causada por cada aplicación.

interfiere. Esto último se debe, entre otras causas, a que como se ha comentado su uso intensivo de las unidades de coma flotante de doble precisión deshabilita la capacidad de las unidades SIMT de emitir dos instrucciones por ciclo.

En resumen, los resultados anteriores demuestran que para maximizar las productividad de las GPGPU cuando ejecutan varias aplicaciones concurrentemente, es necesaria la existencia de un planificador que tenga en cuenta los puntos de contención, la presión que ejerce cada tipo de aplicación en esos puntos y el impacto en las prestaciones que esta contención causa en cada aplicación.

CAPÍTULO 5

Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones de este TFG y se comenta el trabajo futuro.

5.1 Conclusiones

En las tarjetas GPGPU de NVIDIA actuales la concurrencia en la ejecución es una característica que solamente se permite a los kernels lanzados por un sólo proceso. Sin embargo, NVIDIA distribuye software que permite que varios procesos utilicen una tarjeta concurrentemente mediante la intermediación del servicio MPS. Cabe esperar por tanto, que el soporte a la ejecución concurrente de kernels mejore en el futuro, lo que abre la puerta al diseño de planificadores que tengan en cuenta la simbiosis entre aplicaciones para incrementar las prestaciones globales. Este TFG se centra en la caracterización de las aplicaciones con el objetivo de desarrollar este tipo de planificadores.

A este respecto, hemos observado que en la distribución del tiempo de ejecución de la mayoría de aplicaciones estudiadas uno de los componentes principales es la espera debida al acceso a datos en la jerarquía de memoria. Además, se ha comprobado que cuando se lanzan varios kernels en paralelo la degradación de las prestaciones se correlaciona con la intensidad en el uso de las estructuras de memoria compartidas de la tarjeta como la cache de L2 o la memoria principal. Estos resultados sugieren que la planificación debe tener en cuenta el uso de estos recursos compartidos y su impacto en las prestaciones individuales de las aplicaciones para mejorar las prestaciones globales y mejorar la utilización de las capacidades computacionales de la GPGPU.

5.2 Trabajo futuro

El trabajo que seguirá al presente TFG profundizará en la caracterización de las aplicaciones con una granularidad más fina. Para la consecución de esta meta se establecerán los siguientes objetivos:

1. Desarrollar uno o varios *microbenchmarks* que permitan causar interferencias controladas sobre distintos recursos, permitiendo observar los efectos de estas interferencias en las prestaciones y utilización de los recursos. Estos microbenchmarks no sólo se centrarán en la interferencia en L2 y memoria principal, sino en recursos internos del SM, tales como la cache L1 de datos.

2. Analizar y cuantificar el impacto de las interferencias causada por los microbenchmarks así como el comportamiento observado.

3. Diseñar un algoritmo de planificación experimental a partir de los resultados obtenidos.

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 2017. (Accessed on 07/03/2017).

[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. Nvidia fermi architecture whitepaper. http://www.nvidia.es/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, September 2009. (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] 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).

[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] 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).

[11] 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).

[12] 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).

[13] NVIDIA Corporation. [solved]relation of elapsed_cycles_sm and kernel execution time in cuda - nvidia developer forums. https://devtalk.nvidia.com/default/topic/949557/-solved-relation-of-elapsed_cycles_sm-and-kernel-execution-time-in-cuda/, August 2016. (Accessed on 07/03/2017).

[14] NVIDIA Corporation. Understanding difference between instructions issued 1 and instructions issued 2 in computeprof (cud - nvidia developer forums. https://devtalk.nvidia.com/default/topic/539023/understanding-difference-between-instructions-issued-1-and-instructions-issued-2-in-c?offset=3, August 2013. (Accessed on 07/03/2017).

[15] NVIDIA Corporation. Profiler :: Cuda toolkit documentation. http://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference-5x, June 2017. (Accessed on 07/03/2017).

[16] NVIDIA Corporation. Profiler :: Cuda toolkit documentation. http://docs.nvidia.com/cuda/profiler-users-guide/index.html#warp-state-nvvp, June 2017. (Accessed on 07/03/2017).

[17] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[18] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[20] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

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

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

[23] 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.

[24] 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.

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

[26] 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.

[27] Mark Harris. Optimizing cuda. *SC07: High Performance Computing With CUDA*, 2007.

[28] Ryan Smith. Gm200 - all graphics, hold the double precision - the nvidia geforce gtx titan x review. http://www.anandtech.com/show/9059/

the-nvidia-geforce-gtx-titan-x-review/2, March 2015. (Accessed on 07/07/2017).

APÉNDICE A

Configuración del sistema

En este apéndice se presenta brevemente como se ha configurado la plataforma experimental.

A.1 Maquina anfitriona

El ordenador que actua como anfitrión de la GPGPU utilizada es el servidor *xpl3.-gap.disca.upv.es* con un SO *16.04.1-Ubuntu kernel 4.10.0-14-generic*. Se accede al mismo mediante el uso de SSH.

A.2 Instalación de CUDA SDK

Para la instalación de CUDA SDK se empezó por seguir las instrucciones indicadas en la web https://developer.nvidia.com/cuda-downloads. Para este caso concreto se pulsaron los siguientes botones en este orden:

1. linux
2. x86_64
3. Ubuntu
4. 16.04
5. deb (network)
6. Download (2.6 KB)

Tras la descarga del fichero se transfirió a *xpl3* mediante el comando scp y se procedió a instalarlo siguiendo las instrucciones de la web, que rezan:

1. `sudo dpkg -i cuda-repo-ubuntu1604_8.0.61-1_amd64.deb`
2. `sudo apt-get update`
3. `sudo apt-get install cuda`

Finalmente se siguen las instrucciones indicadas en la documentación de cuda para la post-instalación, indicadas en el siguiente página: http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#post-installation-actions.

A.3 Instalación de los *benchmark suites*

A.3.1. Rodinia

Para la instalación de Rodinia, se debe de rellenar el formulario localizado en http://lava.cs.virginia.edu/Rodinia/download.htm y podrá descargarse un archivo comprimido con todo el código fuente de Rodinia.

Tras descomprimir el fichero, se deben de realizar algunas modificaciones para poder compilar todas las aplicaciones sin problemas:

1. Editar el common config del Makefile principal para que no exija que la versión de CUDA sea la 5.5 y pueda utilizar la actual.

2. Editar los Makefile para que la compilación no se realice para la arquitectura SM13 ya que esto causará un error fatal en la compilación

A.3.2. SHOC

La *suite* SHOC cuenta con un repositorio git; por tanto, tan solo clonando el repositorio con el comando "`git clone` https://github.com/vetter/shoc.git" se descarga la versión mas actual de SHOC. Para poder compilar los ficheros, se debe eliminar de los Makefile la llamada para que la compilación sea realizada en la arquitectura SM13.

A.3.3. LonestarGPU

Previa a la instalación de LonestarGPU se debe de descargar y compilar la librería CUB mediante este enlace http://nvlabs.github.io/cub/download_cub.html y seguir las instrucciones que indica el README del fichero.

Tras ello, procedemos a descargar LonestarGPU en el siguiente enlace http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu/download y seguir las instrucciones indicadas en el README. Además, se debe modificar la configuración en los include de los Makefile para que la compilación sea realizada para la arquitectura adecuada.

APÉNDICE B

Codigos fuente

En este apéndice se presentan los codigos utilizados para las mediciones realizadas en este trabajo.

B.1 runProf.sh

```
#/bin/bash

#cuenta atras inicio, mostrar  el estado de la tarjea para asegurarnos de que est  correctamente configurada.
sudo nvidia-cuda-mps-control -d
sudo nvidia-smi --auto-boost-default=0
sudo nvidia-smi --compute-mode=1
echo "Starting in 10 seconds"
sleep 5
nvidia-smi
echo "Starting in 5 seconds"
sleep 5

#CONFIGURACIN
export BH_15k2000t="$HOME/Descargas/lonestargpu-2.0/apps/bh/bh 16000 120 0" # barneshut gravity

export QTC_2="$HOME/Descargas/SHOC/shoc/src/cuda/level2/qtclustering/QTC --size 2"

export heartwall="$HOME/Descargas/rodinia_3.1/cuda/heartwall/heartwall $HOME/Descargas/rodinia_3.1/data/heartwall/test.avi 104"

export lavaMD_5="$HOME/Descargas/rodinia_3.1/cuda/lavaMD/lavaMD -boxes1d 5"

export srad_v1_10k="$HOME/Descargas/rodinia_3.1/cuda/srad/srad_v1/srad 10000 0.5 502 458"

export leukocyte_100="$HOME/Descargas/rodinia_3.1/cuda/leukocyte/CUDA/leukocyte $HOME/Descargas/rodinia_3.1/data/leukocyte/testfile.avi 100"

export METRICS="ipc"
#export EVENTS="active_ctas,elapsed_cycles_sm,active_warps"
export PROFILE_METRICS="--metrics $METRICS"
#export PROFILE_EVENTS="--events $EVENTS"
export APP01=$lavaMD_5
export APP02=$heartwall
export NAME1="APP01"
export NAME2="APP02"

export PROFILE_GENERAL="nvprof --concurrent-kernels on --replay-mode disabled"
```

```
36
37 #eliminamos logs anteriores, solo nos interesan los nuevos
38 rm ~/log/*
39 rm caracterizacion/*
40 #rm compare/*
41
42 #se caracterizan los kernel
43 ./caracterizacion.sh > caracterizacion/log.txt 2> caracterizacion/log.err
44 time ./concurrentProfiling.sh
45
46 #matamos el proceso mpi (iniciado en concurrentProfiling)
47 #ps -ef | grep mps-control | grep cuda | awk '{print $2}' > /tmp/mps.tmp
48 #value=$(</tmp/mps.tmp)
49 #sudo kill $value
50
51 echo -ne '\007'
52 }
```

B.2 caracterizacion.sh

```
1 export CARACTERES="--events l2_subp0_total_read_sector_queries,
    l2_subp1_total_read_sector_queries,l2_subp0_read_sector_misses,
    l2_subp1_read_sector_misses,l2_subp0_total_write_sector_queries,
    l2_subp1_total_write_sector_queries,l2_subp0_write_sector_misses,
    l2_subp1_write_sector_misses"
2
3 #export CARACTERES="--metrics stall_inst_fetch,stall_exec_dependency,
    stall_texture,stall_sync,stall_other,stall_constant_memory_dependency,
    stall_pipe_busy,stall_memory_throttle,stall_not_selected,
    stall_memory_dependency"
4
5 echo "BarnesHutt 15k"
6 time $BH_15k2000t
7 time nvprof $CARACTERES -o caracterizacion/bh16k.nvvp $BH_15k2000t
8
9 echo "QTC_2"
10 time $QTC_2
11 time nvprof $CARACTERES -o caracterizacion/qtc2.nvvp $QTC_2
12
13 echo "heartwall"
14 time $heartwall
15 time nvprof $CARACTERES -o caracterizacion/hw104.nvvp $heartwall
```

```
16
17 echo "lavaMD_5"
18 time $lavaMD_5
19 time nvprof $CARACTERES -o caracterizacion/lavaMD5.nvvp $lavaMD_5
20
21 echo "srad_v1_10k"
22 time $srad_v1_10k
23 time nvprof $CARACTERES -o caracterizacion/sradv1_10k.nvvp $srad_v1_10k
24
25 echo "leukocyte_100"
26 time $leukocyte_100
27 time nvprof $CARACTERES -o caracterizacion/leukocyte100.nvvp $leukocyte_100
```

B.3 concurrentProfiling.sh

```
1 time taskset 0x00000001 $APP01 &
2 time taskset 0x00000002 $APP02
3 #esperamos a que concurrentProfiling termine
4 i="dabama1"
5 while [ "$i" != "" ]
6 do
7 i=$(ps -ef | grep /home | grep Descargas | awk '{print $1}')
8 done
```