Influencia de la Memoria en el Rendimiento de una Arquitectura GPGPU

Francisco Muñoz-Martínez y Manuel E. Acacio [1]

Resumen— **La GPU es un dispositivo con gran potencial de rendimiento para la ejecución de aplicaciones de física, medicina, biología, etc.**

Aun así, dista mucho de ser perfecta y uno de los puntos principales donde se produce esa imperfección es en la jerarquía de memoria. La gran cantidad de hilos ejecutándose sobre este dispositivo genera una exponencial cantidad de accesos a memoria, provocando que las cachés y la memoria principal se ahoguen, actuando como cuellos de botella y evitando que la GPU alcance su máximo potencial.

La investigación presentada a continuación realiza un análisis de la jerarquía de memoria, encontrando que el cuello de botella principal que hay que resolver se encuentra en los niveles superiores de la memoria (memoria caché L1 y red de interconexión) y que los niveles inferiores (L2 y memoria principal) tienen una menor repercusión en este problema. Además, analiza los algoritmos de planificación asentando las bases de lo que puede ser un planificador inteligente que se adapte a la realidad de cada aplicación y que potencialmente podría conseguir mejoras de hasta un 64% en el rendimiento de algunas aplicaciones.

Finalmente, este artículo, utilizando todos los datos obtenidos durante la investigación, plantea nuevos horizontes en la mejora de la jeraquía de memoria de las GPUs, como la incorporación de protocolos de coherencia en las cachés L1 o las técnicas de prefetching para intentar mejorar la tasa de aciertos del nivel congestionado.

Palabras clave— **GPU, warp, jerarquía de memoria, planificación, IPC**

I. Introducción

La GPU (Graphical Processing Unit) es un dispositivo en auge gracias a las grandes capacidades de rendimiento que ofrece respecto a una CPU tradicional. Aplicaciones con alto paralelismo de datos (SIMD) pueden lograr aceleraciones increíblemente grandes sólamente siendo migradas a la GPU. Ésto implica que programas adaptados para la biología, matemáticas, física, medicina, y en general, programas necesarios para el avance científico, puedan ser ejecutados mucho más rápido que con una CPU convencional. En definitiva, estos tipos de aceleradores son actualmente uno de los caballos principales de la ciencia por lo que debemos conseguir que alcancen su máximo potencial.

La GPU por naturaleza está preparada para ejecutar multitud de operaciones simultáneamente. Con el lanzamiento de CUDA en el año 2007, y por consecuencia el verdadero nacimiento de GPGPU (General Purpose Graphical Processing Unit) muchas de las aplicaciones utilizadas por científicos de todo el mundo (matemáticos, físicos, biólogos, etc) han sido migradas para ejecutarse en estos dispositivos, obteniendo un rendimiento muchísimo mayor que con la utilización de una CPU tradicional.

CUDA ofrece un modelo de programación y ejecución sencillo, donde con unas pocas extensiones en C, un programador puede definir y organizar miles de hilos para que éstos se ejecuten masivamente y en paralelo en el hardware de la GPU [1]. El problema fundamental que surge con algunas de las aplicaciones de este tipo, proviene precisamente de tener una gran cantidad de hilos ejecutándose al mismo tiempo, ya que la jerarquía de memoria actual de una GPU encuentra dificultades para absorber el gran número de peticiones de memoria simultáneas que se pueden producir. Este hecho provoca que la memoria actúe como un cuello de botella, provocando paradas en el pipeline, y en consecuencia, afectando muy significativamente al rendimiento de las aplicaciones.

Aunque al principio este problema no era muy significativo debido a que las aplicaciones que se ejecutaban eran de tipo streaming (no reutilizaban datos), poco a poco, conforme se han ido migrando aplicaciones de otros tipos a la GPU, se ha ido acentuando, llegando a ser totalmente fundamental ya que las actuales aplicaciones son cada vez más de propósito general y hacen uso intensivo de la memoria. Por ello, la mejora de la jerarquía de memoria en la GPU es fundamental y probablemente marcará la evolución de este tipo de dispositivos en un futuro cercano.

En este artículo se realiza un estudio sobre el efecto que tiene la jerarquía de memoria en la ejecución de aplicaciones sobre la GPU, analizando diversos parámetros del sistema y localizando los cuellos de botella fundamentales que hacen que el IPC medio de una aplicación ejecutándose sobre una jerarquía de memoria tradicional caiga en un 49,3% respecto a su ejecución sobre una jerarquía de memoria perfecta. Además, también se analiza el planificador que se encarga de emitir instrucciones, ya que como veremos en secciones posteriores, está íntimamente relacionado con la jerarquía de memoria, y su mejora puede traducirse en un aumento de rendimiento para algunas de las aplicaciones.

A continuación, en la Sección 2, se explica el funcionamiento del hardware de una GPU, enfocándose principalmente en su jerarquía de memoria. La Sección 3 asienta el entorno de evaluación que se utilizará para comparar cada experimento realizado y presenta la lista de benchmarks utilizados. Finalmente la Sección 4 muestra todas las pruebas realizadas y los resultados obtenidos que dan pie a las conclusiones en la Sección 5.

[1]Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, e-mail: {francisco.munoz2, meacacio}@um.es

II. La jerarquía de memoria de una GPGPU

La GPU es un dispositivo de cómputo perfectamente preparado para aprovechar el paralelismo SIMD (Single Instruction Multiple Data) de las aplicaciones. Consta de varios multiprocesadores (Streaming Multiprocessor, SM), los cuales se encargan de ejecutar múltiples bloques de hilos utilizando varias unidades de cómputo (Streaming Processor, SP). La unidad mínima que un SM utiliza para ejecutar los diversos hilos que tiene a su cargo es el warp, que consiste en la agrupación de varios hilos en una única instrucción. Es lo que se conoce como paralelismo SIMT [2].

Para la ejecución de los warps, cada SM posee un pipeline de 5 etapas principales:

- Fetch: Un planificador escoge un warp para que se lea la instrucción correspondiente, accediendo a una memoria caché de instrucciones.
- Decode: Se decodifica la instrucción del warp escogido en la etapa anterior y se guarda en un buffer de instrucciones.
- Issue: Uno o varios planificadores escogen una instrucción válida del buffer de instrucciones y la pasan a la siguiente etapa.
- Execution: En esta etapa se ejecuta la instrucción o se accede a memoria, según corresponda su código de operación.
- Write back: Se finaliza la ejecución de la instrucción, liberando y actualizando los recursos necesarios.

Tradicionalmente, la GPU no poseía ningún tipo de jerarquía de memoria ya que se tenía la falsa creencia de que no era necesario reducir las latencias de acceso a memoria pues éstas podían ocultarse perfectamente solapando los accesos a memoria de unos hilos con la ejecución de otros. Ésto depende directamente del algoritmo de planificación que se utilice a la hora de lanzar a ejecutar las instrucciones de los warps en la etapa issue. Lo ideal es un algoritmo que solape perfectamente ejecución y acceso a memoria, es decir, que primero lanze un número determinado de instrucciones de memoria a resolverse y justamente después ejecute las instrucciones de cálculo necesarias hasta que las peticiones de memoria se hayan resuelto para volver a empezar.

Dos algoritmos de planificación bastante usados en los planificadores de la etapa issue son los siguientes:

- priorID (gto): Este algoritmo selecciona la siguiente instrucción del primer warp disponible que entró al buffer de instrucciones.
- Round Robin (rr): Es el clásico algoritmo en donde los warps se van seleccionando consecutivamente de manera circular. Si un warp no está disponible en un momento determinado se elige el siguiente.

El problema es que las latencias de acceso a memoria son demasiado grandes para que puedan ocultarse por lo que las aplicaciones con bajas intensidades aritméticas se ven muy afectadas por la memoria.

Para intentar solventar este problema, a partir de la arquitectura Fermi (la arquitectura que asumimos en este artículo) se introduce una jerarquía de memoria propiamente dicha. Tal y como se puede apreciar en la Figura 1, se dota a cada SM con una memoria caché L1 privada intentando aprovechar la localidad espacial y temporal de los accesos a memoria, evitando así que vayan todos a memoria. Es necesario destacar que las cachés L1 no son coherentes entre sí. A su vez, se introduce una memoria caché L2, compartida por todos los SM, y dividida en varios bancos (donde a cada banco se le asigna un trozo del espacio de memoria). La forma de unir todas las memorias caché L1 y todos los bancos de L2 es utilizando una red de interconexión de manera que cuando ocurra un fallo en una de las L1, su controlador enviará una petición de lectura o escritura al banco correspondiente de la L2, que a su vez, podrá encontrar un acierto o un fallo. Con este último caso mandará la petición a memoria a través de su canal asociado a memoria principal.

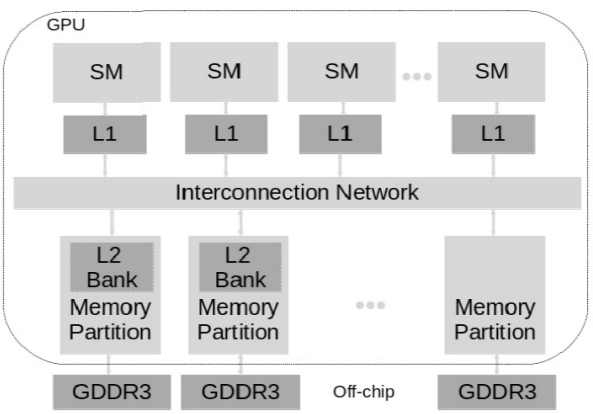

Fig. 1. Esquema del hardware de una GPU indicando los elementos principales.

Cada una de las memorias caché posee una lista de registros especiales denominados MSHRs (Miss Status Holding Registers) que se utilizan cuando un acceso a memoria falla y se produce una petición de acceso al nivel inferior. Así, cuando un warp realice una petición de acceso a memoria a su L1, si ésta falla, el controlador de memoria caché L1 generará una petición de bloque a la L2, manteniendo el estado de la petición en un registro MSHR asociado a la L1. En el momento en el que la L2 resuelva la petición, la L1 guardará el bloque donde corresponda y liberará el MSHR, dejando espacio para futuras peticiones de memoria. Si un warp provoca un fallo de caché y todos los MSHR están ocupados en ese momento, el pipeline se parará hasta que alguno sea liberado, provocando el desaprovechamiento de los recursos del SM que se ha parado.

Debido a que un warp es la agrupación de varios hilos, una instrucción de acceso a memoria provoca por lo general una petición por cada hilo. Para evitar un elevado número de peticiones de memoria se introduce el módulo hardware denominado Memory Address Coalescing entre los SP y la memoria L1 que

se encarga de agrupar varias peticiones consecutivas en una sola, reduciendo así ancho de banda en toda la jerarquía de memoria y ahorrando recursos. El problema es que cuanto más evolucionan las aplicaciones, esta agrupación es más complicada, provocando por tanto un gran número de peticiones en un mismo ciclo, y por tanto, agotando los recursos del pipeline, parándolo, y desaprovechándolo.

III. Entorno de evaluación

Todas las ejecuciones necesarias para nuestro análisis se han realizado utilizando el simulador GPGPU-Sim [3], un simulador de rendimiento a nivel de ciclo que modela una GPU de propósito general y que soporta CUDA [1] y su ISA PTX. Además, es una herramienta flexible que permite variar prácticamente cualquier parámetro del hardware de una GPU. La siguiente Tabla 1 muestra los parámetros del sistema base que se han configurado en el simulador replicando perfectamente una GPU de arquitectura Fermi. A lo largo de esta investigación se utilizará este sistema como punto de referencia a la hora de comparar las distintas pruebas realizadas.

TABLA I
Configuración hardware base

Number of Streaming Multiprocessors	15
Number of Streaming Processors	32
Warp size	32
Number of registers / SM	32K
Number of threads / Core	1024
Number of threads block / Core	8
Shared Memory / Core (KB)	48
Number of Memory Partitions	12
Size L1 Cache (KB)	16
Min L1 Latency (cycles)	1
Number of MSHRs in L1	32
Size L2 Cache (KB)	786
Min L2 Latency (cycles)	120
Number of MSHRs in L2	32
Min DRAM Latency (cycles)	100
Topology Interconnection Network	Fly
Warp Scheduling Policy	RR
Warp Issue Policy	gto
Number of issue schedulers	2

El tamaño de bloque de ambas cachés es de 128 bytes, y ambas siguen la misma política de reemplazo LRU siendo no inclusivas no exclusivas. También, todas siguen la misma política de acierto y fallo en escritura, descrita en la Tabla 2. Vemos que según el tipo de memoria que se mapee en la caché (distinguimos memoria local como la memoria utilizada para la pila y parámetros de funciones) tendremos una política write-back o una en donde cada escritura provoque el desalojo del bloque de la memoria caché.

El análisis de la jerarquía de memoria se ha llevado a cabo ejecutando un conjunto de benchmarks

TABLA II
Políticas de escritura para las cachés L1 y L2

	Local Memory	Global Memory
Write hit	Write-back	evict
Write miss	no-allocate	no-allocate

TABLA III
Clasificación de los distintos benchmarks utilizados para la investigación

Benchmark	Size problem	N Inst.
bfs	1M nodes	481674432
backprop	65536	190054784
cfd	0.2M	5144735592
hotspot	512	110148884
lud-int	256	39890400
needle1	8192	3323187200
srad2	2048x2048	2426994688
streamcluster	32Kx32K	5838454417
convolution2D	4Kx4K	922103954
gemm	512x512x128	379584512
gram	64x65536	2947982080
3d-convolution	512x512x512	10633182270
gaussian	128	54105589
pathfinder	10000000x20	649231040
srad1	520x458	8589033520

de las suites Rodinia 3.1 [4] y Polybench 1.0 [5]. En la Tabla 3 podemos ver todos los benchmarks utilizados, con sus respectivos tamaños de problema y consecuentes número de instrucciones ejecutadas. Todas las aplicaciones excepto cfd, streamcluster y gramm han sido ejecutadas hasta el final.

IV. Resultados

En esta sección vamos a mostrar los resultados de rendimiento de nuestro conjunto de benchmarks ejecutados sobre diferentes configuraciones hardware para ver el impacto de la jerarquía de memoria.

Con el objetivo de ver qué benchmark es más significativo para nuestro estudio, la Figura 2 muestra el porcentaje de instrucciones de memoria (local y global) de cada aplicación (sin hacer distinción entre escrituras y lecturas). Observamos que hotspot y pathfinder tienen únicamente un 1,08% y 1,83% respectivamente, lo que las hace prácticamente insensibles a la memoria. El resto de aplicaciones contienen un porcentaje de instrucciones de memoria superior al 3,5%, siendo más significativas para nuestro estudio, pero destacando sobretodo bfs, streamcluster, convolution 2D, gemm, gram y 3d-convolution con porcentajes superiores al 10%.

A. Sistema base

La Figura 3 muestra el IPC (Instructions Per Cycle) de cada benchmark ejecutado sobre el sistema base descrito en la Tabla I, sobre el mismo sistema pero teniendo una memoria caché L1 perfecta (es decir, un sistema donde cada acceso a memo-

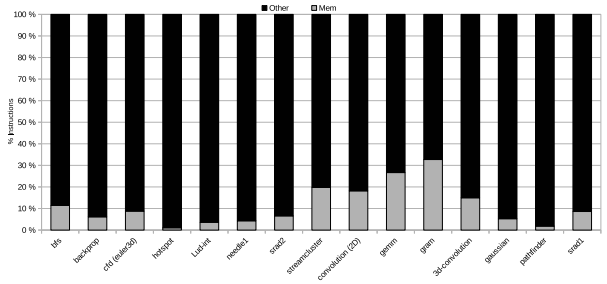

Fig. 2. Porcentaje de instrucciones de memoria de los benchmarks

ria tarda 1 ciclo en resolverse) y un último con la caché L1 desactivada. Con esta comparación vemos que las aplicaciones experimentan un claro aumento de rendimiento al tener una memoria perfecta. De media, las aplicaciones ejecutadas sobre este sistema ideal obtienen un speedup de 2,02 respecto al sistema base. Este hecho se acentúa en las aplicaciones con un alto porcentaje de memoria, llegando incluso a un speedup de 11,81 en streamcluster o 6,11 en bfs. Ésto, unido a que la mejora media de poner la L1 respecto a no ponerla es del 29% (un valor significativo pero pequeño respecto a lo que se puede llegar a alcanzar como puede verse en los resultados del sistema ideal) indica claramente que la jerarquía de memoria tradicional que se está utilizando hoy en día está muy lejos de ser perfecta y que en muchos casos está actuando como un gran cuello de botella que hay que resolver.

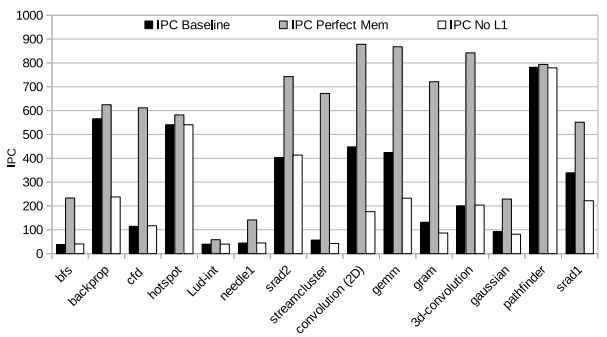

Fig. 3. Resultados de IPC para 3 configuraciones del sistema de memoria

B. Jerarquía de memoria

Con el objetivo de analizar en solitario el funcionamiento de la memoria caché L1 y la red de interconexión se ha modificado el simulador para tener una memoria caché L2 perfecta. Ésto es, que nuestro sistema indica acierto en cada referencia a memoria que vaya dirigido a la L2. Este cambio elimina la congestión en los niveles inferiores de la memoria y nos permite ver el funcionamiento del nivel superior.

Las Figuras 4 y 5 muestran que las aplicaciones con un porcentaje de instrucciones de memoria superiores a un 5% experimentan una reducción media del 11,64% (desde 16 KB a 64 KB) en la tasa de fallos al aumentar el tamaño de la caché L1. Ésto se traduce en un incremento del 15% en el IPC medio. Este hecho indica que en general las aplicaciones tienen localidad temporal pero el reuso de un dato se encuentra muy separado en el tiempo por lo que con tamaños de caché pequeños la información se pierde antes de que pueda ser reusado, aumentando así la tasa de fallos.

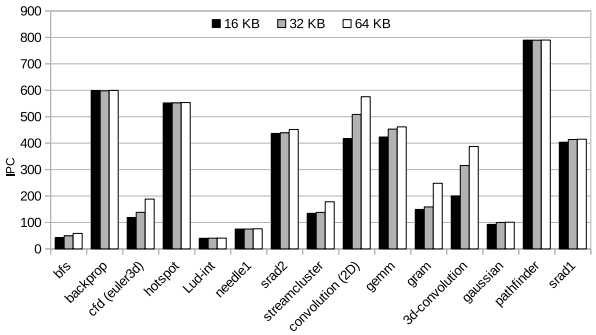

Fig. 4. Resultados de IPC para tamaños de L1 de 16 KB, 32 KB y 64 KB sobre un sistema con L2 perfecta

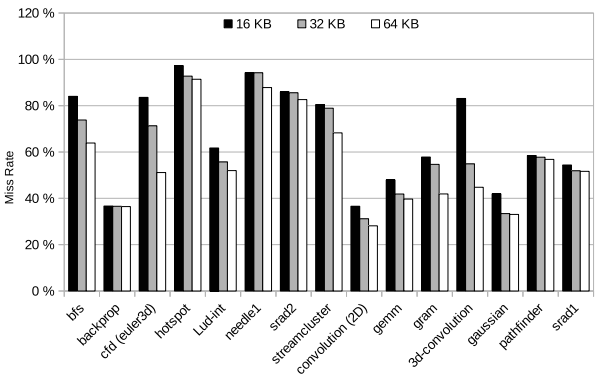

Fig. 5. Resultados de tasa de fallos en la L1 para tamaños de L1 de 16 KB, 32 KB y 64 KB sobre sistema de L2 perfecta

La Figura 6 muestra los resultados de IPC medio y latencia de acceso a memoria media (en ciclos) de 18 ejecuciones realizadas sobre nuestro sistema de L2 perfecta. En estas ejecuciones hemos variado el tamaño de la caché (16 KB, 32 KB, y 64 KB), el número de MSHRs (32, 64, 128), y el número de particiones de memoria (usando 12 y 1024). Podemos ver una clara correlación entre el IPC obtenido y la latencia de acceso a memoria indicando que uno de los problemas fundamentales de la jerarquía de memoria se encuentra en la red de interconexión y en el número de bancos de L2 ya que impacta de manera directa en el rendimiento de las aplicaciones.

De hecho, los picos más altos de IPC (y más bajos de latencia) se encuentran en las pruebas realizadas sobre los sistemas con 1024 particiones de memoria (lo que podemos llamar una red de interconexión perfecta). Al aumentar este número los accesos se reparten entre más puntos, reduciendo los conflictos entre todos los SM, verificando que es totalmente necesario reducir el ancho de banda consumido por la red de interconexión ya que la gran cantidad de accesos a la misma provocan un gran

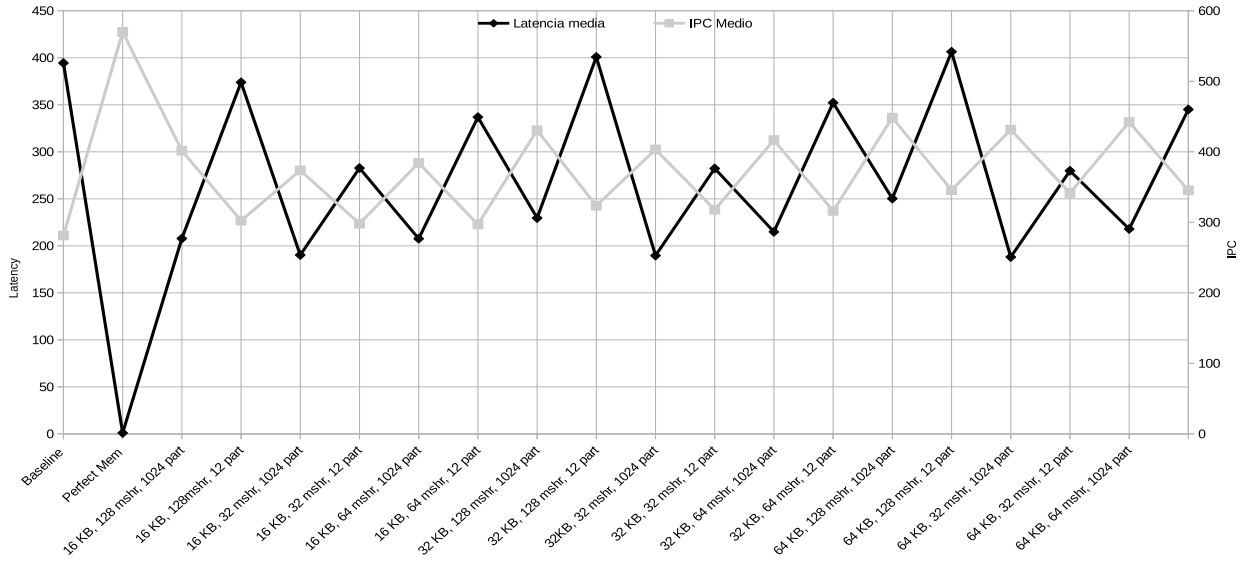

Fig. 6. Correlación IPC medio y latencia media de acceso a memoria sobre 18 configuraciones con L2 perfecta

cuello de botella, disminuyendo el rendimiento de las aplicaciones. Nuestro estudio ha determinado que las pruebas realizadas con 1024 particiones de memoria, es decir, desahogando la red de interconexión, obtienen una mejora media del 25%.

Si aumentamos el tamaño de la caché a 64 KB y la red de interconexión a 1024 particiones de memoria obtenemos una mejora respecto al sistema base con L2 perfecta del 45%. Ya que la reducción de la tasa de fallos de la L1 implica también un ahorro de ancho de banda en la red de interconexión y por tanto un menor número de conflictos entre los SM, el aumentar el tamaño de caché y desahogar la red de interconexión juntos consigue una mejora mayor que la suma de realizar las mejoras individualmente (aumentar la caché mejoraba un 15% y aumentar el número de particiones de memoria un 25% por lo que $15 + 25 = 40$ que es 5 puntos menor que el 45 observado).

Con los resultados de IPC mostrados en la Figura 6 vemos también que aunque el número de MSHRs no es muy significativo en el rendimiento de las aplicaciones (solamente hace variar el IPC entre el 0-4%), el hecho de desahogar la red de interconexión debería de implicar aumentar también este número de manera proporcional para conseguir una mejora media del 4%. Es importante no aumentar el número de MSHRs sin desahogar primero los niveles de memoria inferiores pues puede provocar un mayor número de conflictos en la red de interconexión y por tanto una penalización en el IPC final.

La Figura 7 aglutina los datos de IPC de las aplicaciones ejecutadas sobre el sistema base (ya visto en figuras anteriores) junto con los datos de IPC sobre ese mismo sistema sin memoria caché L2 y con la memoria caché L2 perfecta. Podemos ver que el impacto de la L2 en la jerarquía de memoria es muy pequeño respecto a todo lo que se puede llegar a alcanzar, pues el tener una L2 perfecta apenas implica una mejora media del 5% respecto al sistema

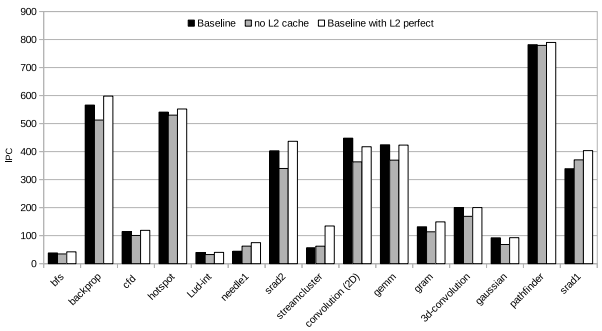

Fig. 7. Datos de IPCs obtenidos en el sistema base con L2, sin L2 y con L2 perfecta

base. Ya que el valor de IPC medio con L1 perfecta (memoria ideal) es de 569 y el obtenido con L2 perfecta es de 298 (y además este último valor es cercano al IPC base de 281) podemos afirmar que el cuello de botella se encuentra en los niveles superiores de la jerarquía de memoria (memoria caché L1 y red de interconexión), y en la contención producida debido al escaso número de bancos de caché L2.

Además, como en el caso de la aplicación convolution 2D, puede ocurrir que al mejorar la L2 (en este caso haciéndola perfecta), el rendimiento de la aplicación empeore ya que se aumenta la congestión en la L1 y en la red de interconexión, provocando más paradas en el pipeline.

C. Algoritmo de planificación

Como vimos en la Sección 2, el hecho de solapar cómputo con accesos a memoria es muy importante en este tipo de arquitecturas y la base para conseguirlo se encuentra en el algoritmo de planificación. Dos de los más usados son los algoritmos Round Robin y priorID por lo que vamos a ver el impacto que tiene utilizar uno u otro en el rendimiento de las aplicaciones.

La Figura 8 muestra el IPC de cada benchmark

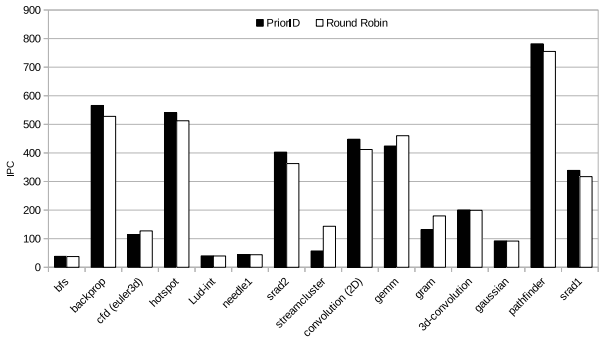

Fig. 8. Resultados de IPC sobre sistemas con planificación priorID y Round Robin

ejecutado con cada uno de los algoritmos de planificación nombrados arriba. De media, ambos algoritmos de planificación obtienen un IPC de 280 pero en concreto hay algunas aplicaciones que funcionan mejor con priorID y otras con round robin.

Con un análisis de las Figuras 2 y 4 (en la columna de los 16 KB), podemos aglutinar todas las aplicaciones que funcionan mejor con el algoritmo Round Robin como aquellas que poseen más de 19% de instrucciones de memoria y cuya tasa de fallos es superior al 45%. Éstas son en concreto streamcluster, gemm y gram. El resto consiguen mejor rendimiento utilizando el algoritmo priorID. Esta mejora al utilizar un algoritmo de planificación u otro se debe a que según la naturaleza de la aplicación actuando de una u otra forma podemos reducir la tasa de fallos en la caché L1, consiguiendo así descongestionar los niveles inferiores de la jerarquía de memoria y al mismo tiempo solapar de una manera más efectiva accesos a memoria y cálculo al reducir las paradas del pipeline.

Las aplicaciones en las que funciona bien el algoritmo Round Robin son aplicaciones de tipo streaming (de ahí su alta tasa de fallos y su gran número de accesos a memoria). Este algoritmo lanza los warps consecutivamente, aumentando el balanceo de los warps y siguiendo el patrón perfecto que por naturaleza este tipo de benchmarks necesita. El algoritmo priorID funciona bien en aplicaciones donde se haga una reutilización mayor de los datos (de ahí que la tasa de fallos de dichas aplicaciones sea menor) ya que siempre intenta ejecutar un warp que ha ejecutado hace poco en el marco temporal.

Es factible conseguir un sistema que vía hardware adapte el algoritmo de planificación al tipo de aplicación utilizando los porcentajes descritos arriba. Este algoritmo adaptativo podría alcanzar un aumento de rendimiento del 64,6% respecto a nuestro sistema base en aquellas aplicaciones de tipo streaming, mientras que el resto seguirían funcionando de la misma manera.

V. Conclusiones

En este trabajo de investigación hemos podido ver un análisis exhaustivo de la jerarquía de memoria que puede ser la semilla de una importante mejora en las GPUs del futuro.

Aunque es un dispositivo complejo, se ha comprobado que un algoritmo de planificación efectivo puede mejorar sustancialmente el rendimiento de las aplicaciones. Bastaría con que el planificador implementado en hardware fuese lo suficientemente inteligente para adaptar el algoritmo en función de la naturaleza de cada aplicación. También, un algoritmo inteligente capaz de predecir lo suficiente para alternar a la perfección cómputo y accesos a memoria obtendría resultados excepcionales.

Por otro lado hemos visto que la jerarquía de memoria de una GPU se encuentra muy lejos de ser perfecta y que está actuando como un total cuello de botella que estrangula todo el pipeline. Es de suma importancia saber que la congestión principal se encuentra en la memoria caché L1, en la red de interconexión que se usa para conectar la L1 con la L2, y el número de bancos de L2, ya que un mayor número permiten resolver más fallos de L1 a la vez.

La clave principal se encuentra en mejorar las comunicaciones entre las L1 de todos los Streaming Multiprocessors, de tal forma que se reduzcan los accesos a la red de interconexión al mismo tiempo que se mejoren las tasas de acierto. Como se indica en [6] muchos de los datos que se solicitan a memoria son innecesarios pues en un mismo instante es muy probable que dicho dato se encuentre en cualquiera de las otras L1 debido al tipo de apliciones con las que se tratamos.

En conclusión, debemos centrarnos en mejorar el algoritmo planificación y posiblemente implementar mecanismos de prefetching y coherencia entre todas las cachés L1. Haciendo esto consiguiríamos acercarnos al valor de rendimiento con memoria ideal que tanto anhelamos.

Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Economía y Competitividad (MINECO) y la Comisión Europea FEDER mediante el proyecto “TIN2015-66972-C5-3-R”.

Referencias

[1] “Nvidia cuda webpage,” http://nvidia.com/cuda.
[2] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym, “Nvidia tesla: A unified graphics and computing architecture,” *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Apr. 2008.
[3] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” in *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 163–174.
[4] 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 Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, vol. IISWC, pp. 44–54, Oct. 2009.
[5] “Polybench webpage,” http://web.cs.ucla.edu/ pouchet/software/polybench/.
[6] Saumay Dublish, Vijay Nagarajan, and Nigel Topham, “Cooperative caching for gpus,” *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 4, pp. 39:1–39:25, Dec. 2016.