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

Caracterización del sistema de memoria de una GPGPU

Trabajo Fin de Master

Máster en Ingeniería de Computadores

Autor: Francisco Candel Margaix

Directores: Salvador Petit y
Julio Sahuquillo

5 de septiembre de 2014

Resumen

La constante necesidad de aumentar la capacidad de cómputo y reducir el consumo energético de los procesadores ha llevado a los fabricantes a diseñar sistemas heterogéneos, que incluyen la GPU en el mismo chip que la CPU. Desde el punto de vista computacional las GPUs son mucho más eficientes en la ejecución de aplicaciones de tipo *streaming*. Además, no incluyen lógica para soportar la ejecución de instrucciones fuera de orden, por lo que son más eficientes desde el punto de vista energético. Para explotar al máximo el rendimiento en cada tipo de arquitectura (GPU y CPU), es necesario poder distribuir el cómputo eficientemente entre ambas plataformas. Un aspecto de especial importancia en la programación de sistemas heterogéneos es cómo se comparte la memoria entre las distintas arquitecturas. En este contexto se plantean nuevos retos de investigación en el soporte de la coherencia en la jerarquía de memoria. Los protocolos de coherencia actuales están principalmente orientados a aplicaciones CPU y requieren adaptarse o reemplazarse por por nuevos protocolos que tengan en cuenta las características particulares de los patrones de acceso de las aplicaciones GPU.

En este trabajo se presenta un estudio sobre el comportamiento de varios protocolos de coherencia para GPUs, uno comercial y otro académico. Se analizan las prestaciones de ambos protocolos variando el número de accesos a memoria en vuelo soportados. Los resultados muestran que el mejor protocolo depende del tipo de aplicación y el número de peticiones soportado.

Palabras clave: GPGPU, GPU, Sistemas heterogéneos, OpenCL, Subsistema de memoria, caches, Directorios, Protocolos de coherencia.

Índice general

Capítulo 1

Introducción

1.1. Buscando el máximo rendimiento

Hasta hace poco tiempo la forma más directa de obtener más prestaciones en el procesador era incrementar la frecuencia. Sin embargo, al aumentar la frecuencia se incrementa el consumo y esto hace que la densidad energética sea cada vez mayor, lo que conlleva elevados costes en la fabricación y refrigeración del chip. Por dicho motivo, la frecuencia del procesador no tiene una tendencia creciente actualmente.

Otra forma que históricamente se ha utilizado para incrementar el rendimiento de los procesadores ha sido incrementar su complejidad y segmentar los *pipelines*, aumentando el número de transistores que podemos encontrar en un chip. Esto se podría continuar haciendose mientras se pueda reducir el tamaño de los transistores, pero requiere mucha inversión, tanto en tiempo como económica, y cada vez es más difícil debido a que los transistores están cerca del mínimo tamaño que se puede implementar en el silicio.

Esto hace necesario la utilización de arquitecturas alternativas especializadas y su incorporación a procesadores de propósito general. Así, cobra fuerza la idea de procesadores heterogéneos donde se dispone de núcleos con arquitecturas distintas, cada uno de ellos especializado en un tipo de cómputo.

1.2. Sistemas heterogéneos

Los sistemas heterogéneos son aquellos donde podemos encontrar procesadores con distintas arquitecturas en el mismo chip. Este tipo de sistemas busca la máxima eficiencia tanto en velocidad de ejecución como en consumo energético, utilizando cada uno de los procesadores de los que dispone

para la ejecución de los distintos tipos de código existentes en un programa o conjunto de programas, de manera que cada procesador ejecuta el código en el que está especializado, ya sea para acelerar la ejecución o de reducir el consumo.

La idea de combinar diferentes arquitecturas en el mismo procesador no es nueva pero recientemente la industria ha puesto su atención en este tipo de sistemas, ya que pueden conseguir un rendimiento similar al de un chip multiprocesador o CMP mientras se minimiza el consumo y la complejidad del hardware.

Recientemente con la creación de la Fundación HSA (Heterogeneous System Architecture) por parte de AMD y otras empresas del sector tecnológico, se ha impulsado la investigación sobre procesadores heterogéneos con sistemas de memoria compartidos que permiten acceso hUMA (heterogeneous Uniform Memory Access) [11] en sistemas con CPU y GPU integrados en el mismo procesador.

La investigación actual se está centrando en el subsistema de memoria debido a que es uno de los principales cuellos de botella de prestaciones, especialmente cuando introducimos en el mismo chip arquitecturas con distintos patrones de acceso a memoria y con requerimientos de ancho de banda diferentes.

Este trabajo se centra en la caracterización y mejora del subsistema de memoria de las GPGPU. Para ello, utilizamos el simulador Multi2Sim, que nos permite implementar distintas arquitecturas de forma detallada. Los estudios se centrarán en la arquitectura comercial Southern-Islands (SI) de AMD y en el protocolo académico NMOESI.

1.3. ¿Qué es una GPGPU?

Una GPGPU, siglas de General-Purpose computing on Graphics Processing Units, es un conjunto de técnicas que incorporan las GPUs para acelerar aplicaciones de propósito general, aprovechando el paralelismo ofrecido por las GPUs. Este conjunto de técnicas, afectan, entre otras partes del sistema, al pipeline de las GPUs. Esta evolución de las GPUs convencionales permite a los programadores la posibilidad de desarrollar *kernels* que no tengan como objetivo exclusivo el procesamiento multimedia, sino la ejecución de cualquier tipo de algoritmo masivamente paralelo.

Actualmente, para programar aplicaciones GPGPU existen dos alternativas principales:

- CUDA: plataforma desarrollada exclusivamente por Nvidia para permitir programar aplicaciones para sus tarjetas.

- OpenCL: desarrollado por Khronos group, un consorcio sin ánimo de lucro responsable también del estándar OpenGL.

En los últimos años, debido al rendimiento potencial que las GPUs pueden ofrecer se ha producido un auge en el uso de estos sistemas, en constante evolución. Sin embargo, para obtener el máximo potencial, es necesario identificar y resolver los problemas de prestaciones que estos sistemas todavía adolecen.

1.4. Los protocolos de coherencia en los procesadores heterogéneos

Las CPUs convencionales disponen de complejos protocolos para garantizar la coherencia entre las instrucciones de acceso a memoria de los distintos hilos de ejecución en las aplicaciones paralelas. Estos protocolos no se han implementado hasta ahora en las GPUs, debido a que históricamente se han dedicado a la ejecución de aplicaciones multimedia, cuyos patrones de acceso a memoria no requieren asegurar la coherencia. Además, hasta ahora, los sistemas heterogéneos CPU/GPU han particionado la memoria física disponible, asignado una parte a la GPU y otra a la CPU, de manera que cada parte sólo puede ser accedida eficientemente por una de las dos arquitecturas. En este entorno, no hay comunicación entre la CPU y la GPU durante la ejecución de cargas heterogéneas, limitándose ésta a la copia de datos de entrada y resultados antes y después respectivamente de la ejecución de los kernel GPGPU.

Sin embargo, las nuevas APUs (Accelerated Processing Unit) de última generación, además de incorporar una CPU y una GPU en el mismo chip, implementan arquitecturas del sistema de memoria denominadas hUMA. En este tipo de arquitecturas, el acceso a la memoria está íntegramente compartido entre la CPU y la GPU, sin que exista un particionado *a priori*. El objetivo es que las aplicaciones heterogéneas CPU/GPU puedan compartir datos mediante la jerarquía de memoria durante la ejecución de la carga heterogénea sin necesidad de réplicas. En este contexto, las prestaciones del protocolo de coherencia cobran especial relevancia, ya que este protocolo tiene que llevar a cabo la coherencia entre aplicaciones con patrones de acceso muy distintos y uso de ancho de banda diferentes, como son las aplicaciones GPGPU y CPU.

Por tanto, se requiere de un estudio detallado del comportamiento respecto al acceso a memoria de las aplicaciones GPGPU a fin de comprender causas de pérdidas de prestaciones y proponer mejoras para los protocolos

de coherencia existentes actualmente en este tipo de sistemas.

1.5. Objetivos

En líneas generales, los principales objetivos que persigue este Trabajo Final de Máster son:

1. Entender las ventajas e inconvenientes de la arquitectura de memoria de las GPUs frente a la de las CPUs, así como sus principales particularidades en cuanto a la metodología de programación y patrón de acceso a memoria.

2. Implementar un sistema de memoria GPU actual, ya su diseño está en constante evolución y se trata de un tema de investigación candente. En concreto, se modelará en Multi2Sim la jerarquía incluida en las tarjetas gráficas de AMD Southern Islands.

3. Estudiar el impacto de los protocolos de coherencia en las prestaciones de las GPUs. En concreto, se estudiará como los protocolos MOESI y NMOESI afectan a las prestaciones del sistema de memoria modelado.

4. Identificar y proponer alternativas de mejora para los sistemas de memoria y protocolos de coherencia en las GPUs.

El resto de esta disertación se organiza como sigue. En el capítulo 2 se presenta el trabajo relacionado. En los capítulos 3 y 4 se describen la plataforma de programación OpenCL y la arquitectura GCN, respectivamente. En el capítulo 5 se analiza en profundidad las características de los sistemas de memoria de las GPU, en concreto el de la familia Southern Islands. En el capítulo 6 se presenta el simulador con el que se ha realizado este estudio. En el capítulo 7 se describen los experimentos realizados y se discuten sus resultados. Por último, en el capítulo 8 se presentan las conclusiones y el trabajo futuro.

Capítulo 2

Trabajo relacionado

Existen notables diferencias entre las CPUs y las GPUs que hacen que cada arquitectura esté especializada en un tipo distinto de cómputo. Las características principales que diferencian a las GPUs son:

- Tolerancia a altas latencias de acceso a memoria.
- Gran cantidad de *threads* (del orden de miles) ejecutándose simultáneamente.
- Ausencia de protocolo de coherencia en el sistema de memoria, recayendo la responsabilidad de la coherencia sobre el programador.

Debido a estas diferencias, es necesario adaptar de forma adecuada las técnicas clásicas, usadas en las CPUs, para mejorar el rendimiento del sistema de memoria en las GPUs. Hasta ahora, la investigación se ha centrado en tres grandes aspectos: memorias cache, protocolos de coherencia, y *programabilidad*.

Un estudio interesante sobre la conveniencia de incluir memorias cache típicas como las de las CPU se presenta en [2]. Este estudio concluye que algunas aplicaciones mejoran sus prestaciones con las memorias cache, mientras en otras se observa que sus prestaciones pueden decaer ostensiblemente. Una posible solución sería diseñar memorias cache que funcionen de forma adaptativa y que se habiliten/deshabiliten en función de las características de la carga.

También hay que considerar que debido al rápido incremento en número de transistores en las GPUs, se puede llegar a un excesivo consumo energético. Por eso en [12] se propone el uso de caches de pequeño tamaño por encima de L1 para cada elemento de proceso o PE de cada núcleo de la GPU, consiguiendo reducir un 35 % el consumo del sistema de memoria.

Respecto a los protocolos de coherencia, es sabido que estos necesitan adaptarse al tipo de cómputo masivamente paralelo que presentan las GPUs. Este tipo de protocolos fueron originalmente diseñados en entornos con relativamente pocos fallos de cache en vuelo. Por tanto, cuando este tipo de protocolos se aplican a entornos con miles de peticiones en vuelo como en las GPUs, se saturan con facilidad, con la consiguiente pérdida de prestaciones. En el artículo [5] se presenta un protocolo de coherencia de directorio híbrido que permite evitar que lo accesos no-coherentes saturen el directorio.

En [6], aparte de compararse distintos protocolos de coherencia, se presenta un protocolo para implementar coherencia en los sistemas de memoria de las GPUs sin la necesidad de directorio, que es uno de los principales cuellos de botella en la implementación de protocolos de coherencia para GPUs.

El orden en el que se procesan la gran cantidad de transacciones de memoria en las GPUs también puede afectar notablemente al rendimiento. En [8] se presenta una técnica para reordenar los accesos a memoria desde el primer nivel de cache. En [13] también se investiga la reordenación de peticiones, pero centrándose en L2 y la memoria principal.

Nótese que el modelo de consistencia que soporta el protocolo de coherencia tiene repercusión en la facilidad de programación o programabilidad de las aplicaciones GPGPU. Si el modelo de consistencia se relaja, es posible reducir la complejidad y el impacto en las prestaciones de la implementación del protocolo de coherencia, a costa de hacer menos intuitiva la programación de las aplicaciones. Recientemente, en [1] se comparan distintas implementaciones de varios modelos de consistencia, y se concluye que el modelo usado en las GPUs no afecta notablemente a las prestaciones, pudiendo éste ser elegido basándose en otros parámetros, como la complejidad del hardware, la eficiencia energética, o la programabilidad, en lugar de en función de la productividad o throughput.

Por otro lado, para mejorar la programabilidad, se han propuesto técnicas que permitan la extensión de la memoria disponible en las GPU discretas, a la vez que se intenta hacer más trasparente la comunicación entre CPUs y GPUs en sistemas heterogéneos. En este sentido surgen propuestas como RSVM [10] o la arquitectura HSA [14] para combinar los sistemas de memoria de ambos tipos de procesadores.

En particular, la arquitectura HSA se ha implementado en algunos chips de AMD. Algunos estudios demuestran que combinar ambas arquitecturas en el mismo chip se puede conseguir un rendimiento igual o incluso superior a procesadores mucho más complejos [15], ya que las GPUs en el chip permiten acelerar en gran medida una parte importante de las aplicaciones.

Finalmente, no todas los propuestas para mejorar las prestaciones de las GPUs se centran en el sistema de memoria. Otros trabajos investigan aspectos

diversos como delimitar el tamaño óptimo para los wavefront [16], agrupar los threads para que sea más fácil controlar las divergencias de código [18, 17] o maximizar la utilización del hardware [9].

Capítulo 3

OpenCL

OpenCL, del inglés Open Computing Language, surge de la convergencia de varias empresas de sector tecnológico ante la necesidad de crear un estándar multiplataforma que permita aprovechar al máximo la tecnología actual.

Debido a las necesidades específicas de los sistemas heterogéneos, se precisaba de un entorno software para interactuar con las distintas arquitecturas presentes en el sistema y especificar qué arquitectura debe ejecutar cada parte de la carga. Además, hacía falta desarrollar métodos y técnicas para poder expresar el cómputo masivamente paralelo de forma sencilla y eficaz. En un principio, AMD propuso el uso de Brook, un lenguaje desarrollado en la Universidad de Standford para controlar la computación de *streams* (streaming computations). Además, AMD desarrolló las plataformas Close To Metal (CTM) y la Compute Abstraction Layer (CAL) para permitir a los desarrolladores adaptar sus aplicaciones a cada modelo de GPU comercializado por AMD y optimizar al máximo el rendimiento. Ambas plataformas fueron abandonadas y sustituidas por OpenCL pocos años después de su lanzamiento.

OpenCL que lleva desde 2008 en desarrollo, y ya se encuentra en la versión 2.0. Por otro lado, Nvidia empezó a desarrollar CUDA en 2007, propuesta que en principio tuvo más éxito que los homólogos propuestos por AMD. Actualmente la plataforma CUDA cuenta con una gran aceptación dentro de la programación GPGPU, siendo la principal rival de OpenCL.

La versión más reciente de OpenCL, aparte de permitirnos crear y ejecutar aplicaciones GPGPU (o *kernels* si se sigue la terminología de AMD), añade la posibilidad de compartir el espacio de memoria entre la CPU y la GPU en las APUs de nueva generación con tecnología hUMA, así como la creación de tareas en cascada.

3.1. Modelo de la plataforma

El modelo de plataforma que define OpenCL consiste en un *host* al cual podemos conectar uno o más dispositivos OpenCL. Cada dispositivo consta de una o varias unidades de cómputo denominadas CUs (*Compute Unit*), las cuales se dividen a su vez en varios elementos de proceso o PEs (*Processing Elements*), que son los que finalmente llevan a cabo el cómputo.

Una aplicación OpenCL tiene dos partes: el código del *host* y el código del *kernel*. El primero se ejecuta en el host y se encarga de enviar los kernels a las unidades OpenCL presentes en el sistema, que son las encargadas de procesar los kernels.

Los kernels se programan en OpenCL C, un subconjunto de C99. Cada plataforma que implementa el estándar OpenCL incluye un compilador con el que poder convertir el código fuente en código ejecutable para ser procesado en los dispositivos OpenCL de esa misma plataforma. Los kernels se pueden compilar de dos formas: i) *online* cuando el código del host compila el kernel usando el API que el estándar proporciona; o ii) *offline* previamente a la ejecución mediante el uso de las herramientas disponibles en la plataforma.

3.2. Modelo de ejecución

El modelo de ejecución de OpenCL consta dos unidades de ejecución distintas. Por un lado, el host prepara el contexto OpenCL y lo configura, y por otro lado el kernel realiza la computación masivamente paralela dentro de este contexto. El contexto es definido por el entorno en el que se ejecutan los kernels y se compone de las siguientes partes:

- Dispositivos o unidades de cómputo: uno o más dispositivos que implementan el estándar OpenCL, presentes en el sistema.
- Objetos del kernel: las funciones y sus argumentos, que se ejecutarán en los dispositivos.
- Objetos de la aplicación: el código fuente y el código ejecutable que implementan el kernel.
- Objetos de la memoria: variables accesibles tanto por el host como por los dispositivos OpenCL.

La parte de la aplicación ejecutada por el host usa el API de OpenCL para crear y gestionar el contexto. Este API permite al host interactuar con los dispositivos OpenCL presentes en el sistema usando una cola de comandos.

Cada cola de comandos esta asociada con un sólo dispositivo. Los comandos pueden ser de 3 tipos: comandos para encolar un kernel, comandos para gestionar la memoria entre el host y el dispositivo OpenCL, y comandos de sincronización. Además, un kernel en ejecución también puede encolar otros kernels. Los comandos para encolar kernels pasan por 6 estados: *queued*, *submitted*, *ready*, *running*, *ended*, y *complete*. Los comandos comunican su estado a través de eventos.

Cuando un kernel se ejecuta, se generan varias llamadas a su código, denominadas instancias del kernel o *kernel-instances*. Estas instancias se identifican por su valor del índice *NDRange*, descrito en la siguiente sección. Cada una de las instancias es ejecutada por un thread distinto en el dispositivo. El trabajo de cómputo asociado a cada una de estas instancias se denomina *work-item*. Los work-items son agrupados por los dispositivos OpenCL en *work-groups*.

3.2.1. NDRange

Un NDRange es un índice N-dimensional, donde N puede ser 1, 2, ó 3 dimensiones. El espacio de threads indexado por un NDRange se divide en bloques que representan a los work-groups, que a su vez se componen de work-items. Los work-items representan los threads que ejecutarán las instancias del kernel y que son la menor unidad de trabajo de cómputo en OpenCL. El rango indexado por un NDRange está definido por 3 vectores de enteros con N elementos que indican:

1. El valor de la longitud del espacio indexado en cada dimensión.
2. El valor inicial del índice en cada dimensión. Este valor es 0 por defecto.
3. El valor de la longitud del espacio asignado a un work-group en cada dimensión.

En la figura 3.1 se puede ver un ejemplo donde se definen las dimensiones (en el ejemplo solo 2) del NDRange y de los work-groups antes de lanzar su ejecución. Cada work-item tiene asociado un ID global definido por una tupla N-dimensional dentro del rango indexado por el NDRange y un ID local que indica la posición que ocupa dentro del work-group.

3.3. Modelo de memoria

El modelo de memoria de OpenCL especifica la estructura, contenido y comportamiento de la memoria que encontramos en una plataforma OpenCL.

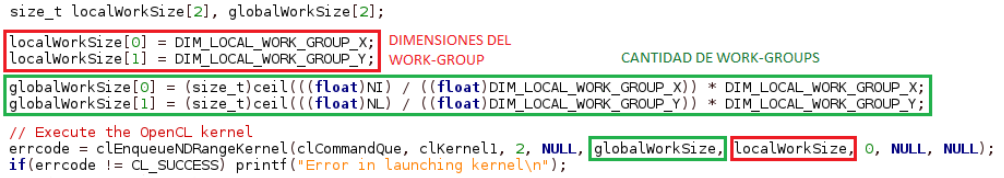


Figura 3.1: Ejemplo de configuración del tamaño de un NDRange y de los work-groups.

Conocer el modelo de memoria, aún sin saber en qué dispositivos será lanzado el kernel, permite que los programadores tengan una visión más clara y sencilla de cómo diseñar el kernel. El modelo de memoria define de forma precisa cómo interactúan todos los dispositivos OpenCL con la memoria presente en un contexto y también cómo lo hacen los work-items dentro de cada dispositivo. El estándar OpenCL define el modelo de memoria en cuatro partes:

- Regiones de memoria: son las distintas particiones de memoria física entre el host y los dispositivos OpenCL.
- Objetos de memoria: son los objetos definidos por la API de OpenCL, accesibles tanto por el host como por los dispositivos OpenCL.
- Memoria virtual compartida: Espacio de memoria virtual direccionable por el host y todos los dispositivos del sistema.
- Modelo de consistencia: define las reglas por las que se rige la coherencia en los accesos a memoria simultáneos, así como las operaciones atómicas y las barreras que permiten ordenar los accesos a memoria y crear puntos de sincronización.

El modelo de memoria de OpenCL segrega la memoria física asignada al host y la de los dispositivos OpenCL. Por tanto, para compartir datos entre los diversos componentes se debe usar o bien la API o el espacio de memoria virtual compartida.

Tal como se muestra en la figura 3.2, la memoria física asignada a un dispositivo OpenCL se compone de cuatro regiones de memoria disjuntas:

- Memoria global: esta región de memoria permite accesos tanto de lectura como de escritura para todos los work-items que ejecutan un kernel. Los accesos a esta región pueden ser *cacheados* dependiendo de la jerarquía de la arquitectura de memoria del dispositivo.

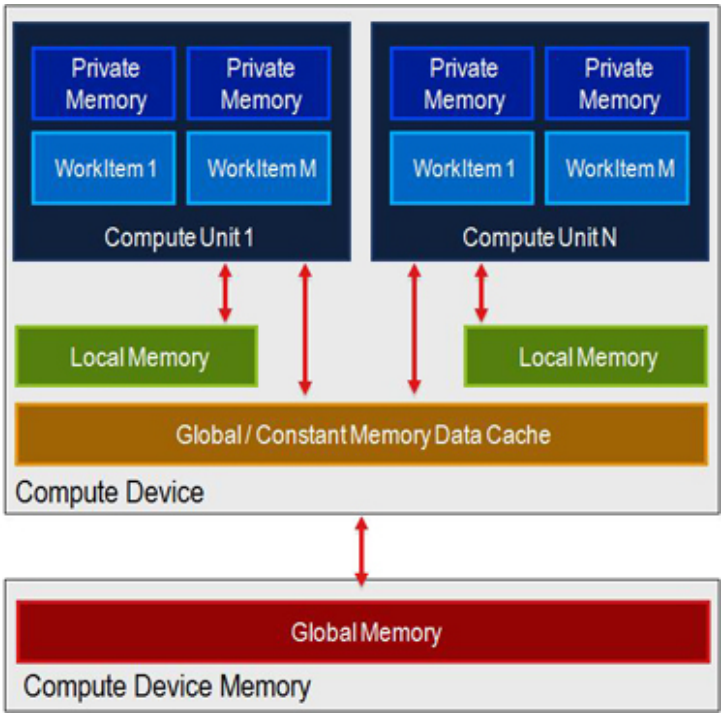


Figura 3.2: Regiones del modelo de memoria de OpenCL.

- Memoria de constantes: esta región es inicializada por el host y sólo permite accesos (*cacheables*) de los work-items.
- Memoria local: esta región es de uso exclusivo para cada work-group y permite que los work-items asociados compartan datos.
- Memoria privada: esta región es privada para cada work-item y está compuesta por las variables definidas de forma privada y con valores potencialmente distintos para cada work-item.

Hay que remarcar que la memoria privada y la local están asociadas siempre al mismo dispositivo OpenCL, mientras que la memoria global y de constantes son compartidas entre todos los dispositivos presentes en el contexto. Aunque no estén presentes en el modelo, la jerarquía de memoria puede incorporar caches para incrementar el rendimiento de los accesos a memoria.

Finalmente, una de la principales mejoras de la versión 2.0 de OpenCL es la incorporación de SVM (Shared Virtual Memory) que permite compartir datos entre el host y los dispositivos OpenCL mediante el espacio de memoria virtual y sin necesidad de realizar copias entre las regiones de memoria física

asignadas a cada tipo de dispositivo. Esta mejora incrementa las prestaciones y hace más eficiente el acceso a los datos en los sistemas heterogéneos.

3.4. Resumen

En esta sección se ha explicado de forma general las partes que forman el estándar OpenCL y su funcionamiento. En el capitulo siguiente se explica como se implementan estos conceptos en el hardware.

Capítulo 4

Arquitectura de una GPGPU

Los principales diseñadores de procesadores gráficos actuales son Nvidia y AMD. Cada una de las empresas nombradas plantea un enfoque distinto sobre la arquitectura de las GPUs. AMD está más centrada en la combinación de la arquitectura de la GPU con la de la CPU, formando así un sistema heterogéneo donde el cómputo sea ejecutado por el procesador más conveniente. Con esta meta, las últimas GPUs de AMD usan un tamaño de bloque en su jerarquía de memoria igual al utilizado en las CPUs con arquitectura x86. Además, incluyen unidades aritmético-lógicas con un comportamiento más simple e intuitivo desde el punto de vista de la computación de propósito general. En este sentido, OpenCL, el lenguaje seleccionado por AMD para la programación de sus tarjetas, se ajusta al objetivo de que sean usadas para este tipo de cómputo.

Por su parte Nvidia se centra sólo en el mercado de las plataformas GPU no heterogéneas. El procesador de sus últimas tarjetas incorpora una arquitectura GPU clásica. Sin embargo, la plataforma de programación de Nvidia denominada CUDA (presentada en 2007) tiene una gran aceptación entre los desarrolladores. Debido a que Nvidia no ha distribuido información específica sobre la arquitectura de su nueva arquitectura SMX, este capítulo se centra en explicar la arquitectura implementada por AMD.

4.1. La arquitectura *Graphics Core Next*

La arquitectura del nucleo GPU más reciente de AMD se denomina *Graphics Core Next* (GCN)[3]. La figura 4.1 presenta un diagrama con los principales componentes de los núcleos computacionales o CUs en esta arquitectura. Esta arquitectura se presentó en 2011 y representa un importante avance en las arquitecturas para GPUs, ya que es una arquitectura más enfocada al

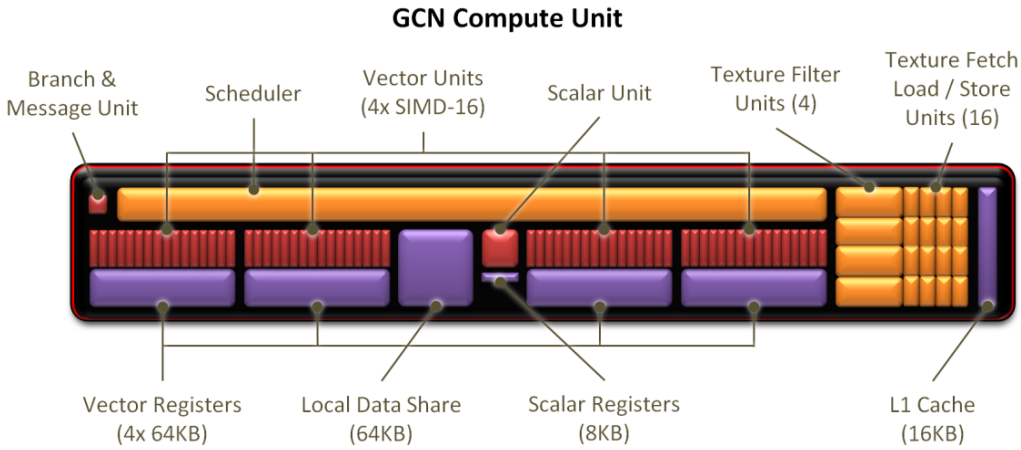


Figura 4.1: Unidad de computo de AMD.

cómputo de propósito general que a las aplicaciones multimedia. Para ello, se rediseñó la arquitectura de las unidades aritmético-lógicas SIMD (Single Instruction Multiple Data) presentes en la GPU, tal como se puede apreciar en la figura 4.2.

La principal modificación es la sustitución de las instrucciones VLIW (*Very Large Instruction Word*), capaces de realizar varias operaciones a la vez, por instrucciones que realizan una única operación. Este cambio se debe a que las últimas son más intuitivas para la optimización del código desde el punto de vista del programador. A este cambio fundamental en la arquitectura del juego de instrucciones se añade una rediseño de las unidades SIMD. En la arquitectura VLIW4 cada unidad SIMD ejecutaba 4 operaciones simultáneamente, mientras que en GCN las unidades SIMD son capaces de ejecutar a la vez instrucciones de 16 threads distintos (es decir, 16 operaciones). Este esquema se denomina SIMT (*Single Instruction Multiple Thread*).

A un conjunto de 64 threads ejecutados por la misma unidad SIMD se le denomina *wavefront*. Puesto que una unidad SIMD sólo puede ejecutar simultáneamente la misma instrucción de 16 threads, un wavefront se divide en 4 *subwavefronts* con 16 threads cada uno. Estos subwavefronts se ejecutan en la misma unidad SIMD multiplexándolos en el tiempo. De esta forma la unidad SIMD se comporta como si ejecutara los 64 threads de un wavefront simultáneamente.

En resumen, GCN presenta una arquitectura más simple mediante la agrupación de la ejecución de varios threads en la misma unidad SIMD. En las arquitecturas anteriores cada SIMD procesaba un thread con el que se podían realizar varias operaciones simultáneamente a través de instrucciones

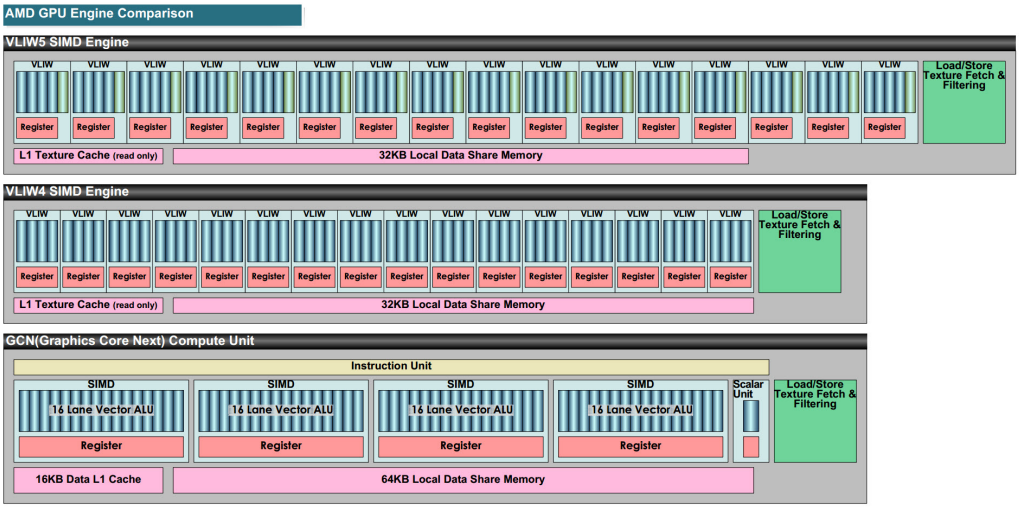


Figura 4.2: Arquitecturas VLIW5, VLIW4, y GCN.

VLIW. Esta filosofía cambia en GCN, donde cada SIMD ejecuta a la vez instrucciones sencillas de 64 threads distintos.

4.1.1. Front-End

Para tolerar operaciones de alta latencia sin perder prestaciones, a cada SIMD se le asigna la ejecución de 10 wavefronts distintos. Para soportar la ejecución de estos 10 wavefronts, el *front-end* del SIMD (figura 4.3) dispone de 10 contadores de programa y buffers de instrucciones distintos. Por otro lado, los SIMD se agrupan de 4 en 4, en núcleos denominados Compute Units (CUs). Por tanto, cada CU puede tener asignada la ejecución de 40 wavefronts. Nótese que cada uno de estos wavefronts puede pertenecer a diferentes work-groups. Puesto que a cada thread, se le asigna un work-item distinto, una GPU que implemente, por ejemplo, 32 CUs podría ejecutar con la arquitectura GCN hasta 81920 work-items simultáneamente.

Cada grupo de 4 CUs comparten una cache L1 de instrucciones de 32 KB, la cual tiene 4 vías y está respaldada por la cache L2. Los bloques de la cache son de 64B. La política de remplazo es LRU, dispone de 4 bancos y puede devolver una instrucción de 32 bits por ciclo para cada una de las 4 CUs.

Una vez se ha buscado la instrucción, esta se almacena en el buffer de instrucciones correspondiente al wavefront, llamado wavefront buffer. Cada SIMD tiene acceso a un *pool* con 10 wavefront buffers. Las siguientes etapas del pipeline decodifican y emiten las instrucciones almacenadas en estos buf-

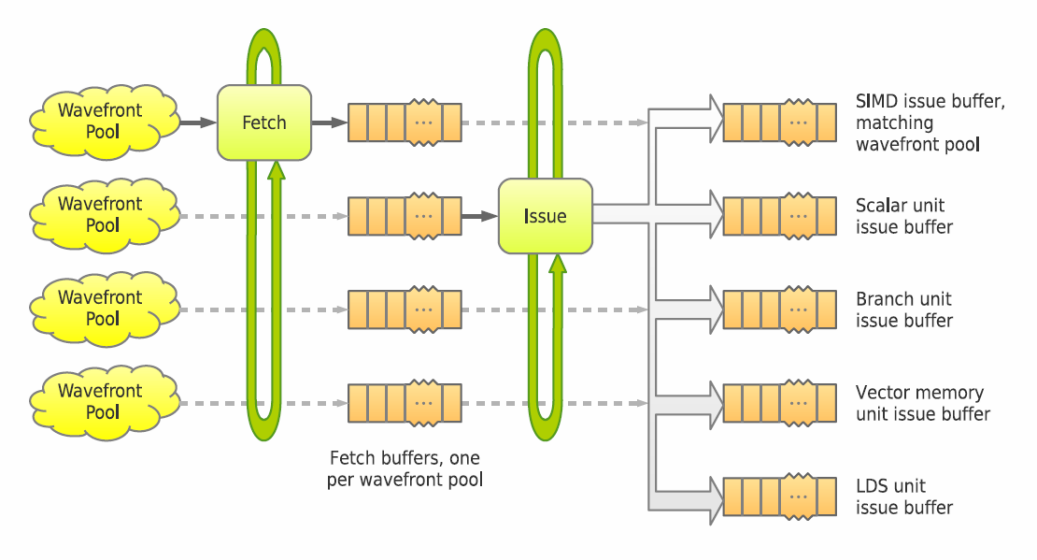


Figura 4.3: Pipeline del front-end.

fers. Para ello, el CU selecciona uno de sus SIMD, usando round-robin, para decodificar y emitir hasta 5 instrucciones cada ciclo. Para preservar la ejecución en orden, las 5 instrucciones deben pertenecer a wavefronts distintos. Además, deben ser de distinto tipo debido a restricciones en el pipeline de los SIMD.

Nótese que puesto que una unidad SIMD multiplexa en el tiempo 4 sub-wavefronts, requiere 4 ciclos para ejecutar una instrucción de un wavefront. Debido a esta latencia, seleccionar sólo uno de los SIMDs del CU para emitir las instrucciones cada ciclo es suficiente para utilizar completamente los recursos computacionales del CU.

El SIMD puede decodificar y emitir hasta 5 instrucciones de los 10 buffers de instrucciones a la unidad de ejecución. Además, una instrucción especial (NOP, barreras, interrupciones, etc.) puede ser ejecutada en el wavefront buffer sin usar ninguna unidad funcional. Cada CU tiene 16 buffers dedicados a seguir las instrucciones de barrera, las cuales fuerzan a que un wavefront se sincronice globalmente.

4.1.2. Unidad escalar

La unidad escalar se encarga tanto de las instrucciones aritmético-lógicas escalares como de acceso a memoria escalar. Esta unidad accede a la jerarquía de memoria usando solamente el espacio de memoria de constantes, por lo que sólo puede realizar operaciones de lectura.

Para acceder al espacio de memoria de constantes, la unidad dispone de una cache de constantes de sólo lectura en L1. Esta cache tiene un tamaño de 16 KB repartidos en 4 vías con un tamaño de línea de 64B y política de remplazo LRU, y consta de 4 bancos con un rendimiento de 16 B/ciclo por banco. Al igual que la cache de instrucciones, esta cache se comparte en grupos de hasta 4 CU.

Además, cada unidad escalar dispone de un banco de registros de 8 KB, dividido en 4 bloques de 2 KB, con 512 registros para cada SIMD. Estos 512 registros se reparten entre los 10 wavefronts asignados a cada SIMD. Como máximo 1 wavefront puede usar 112 registros. Los registros son de 32 bits, pudiéndose almacenar datos de 64 bits usando 2 registros consecutivos.

Una de la tareas principales de esta unidad es la ejecución de las instrucciones de control de flujo, las cuales se ejecutan en cada CU por separado, reduciendo el consumo y aumentando las prestaciones al evitar las grandes latencias y el sobrecoste energético que ocasionaría un control de flujo centralizado.

Para la implementación del control de flujo, la unidad escalar cuenta con dos pipelines. El primero de ellos se dedica al manejo de las interrupciones y algunos tipos de sincronización. El segundo dispone de una unidad aritmético-lógica para operar con enteros de 64 bits. Esta unidad permite acelerar varias operaciones de control de flujo.

4.1.3. Unidad de memoria local (LDS)

Para plataformas orientadas al cómputo com GCN, la comunicación y sincronización son vitales para mantener sus altas prestaciones, especialmente para las aplicaciones de propósito general emergentes. La memoria LDS (memoria compartida para datos locales, figura 4.4) es una memoria de baja latencia con direccionamiento explícito accedida a nivel de work-group.

La memoria LDS tiene una capacidad de 64 KB con 16 ó 32 bancos, dependiendo de la gama del producto. Cada banco contiene 512 entradas de 32 bit. Dispone de 2 puertos de entrada, cada uno de ellos con 16 lineas y asociado a 2 SIMDs. Esta unidad esta preparada para poder escribir simultáneamente 32 lineas cada ciclo. Dispone de un crossbar all-to-all que conectar las 32 lineas de entrada con los 32 bancos de memoria. Normalmente, se combinan las 16 lineas de los 2 puertos de entrada, abarcando los accesos que producen 2 wavefronts cada 4 ciclos. Los conflictos de banco son detectados y solucionados por el hardware en grupos de 32 work-items perteneciente al mismo wavefront.

La LDS puede escribir los datos en los registros vectoriales (vGPRs) o enviarles directamente a los SIMDs. También puede leer datos de los vGPRs

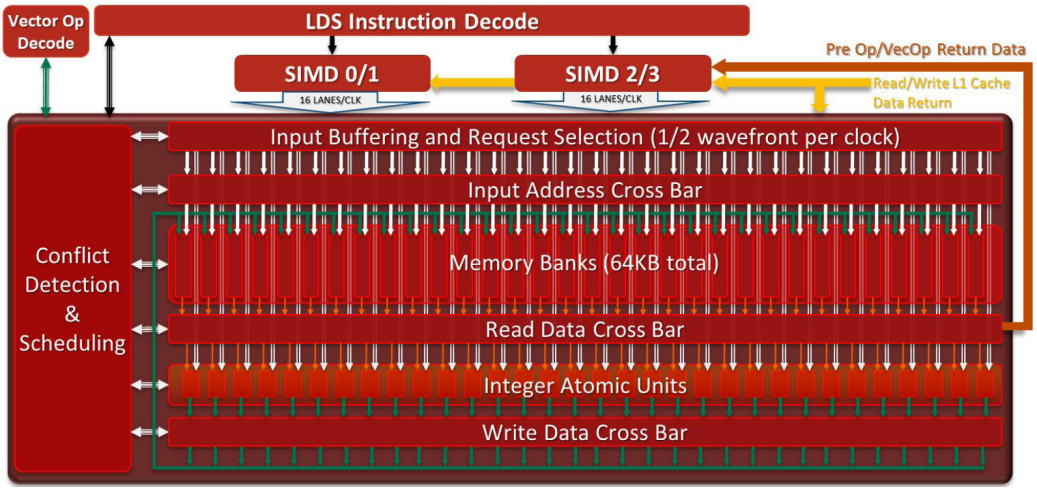


Figura 4.4: Local Data Share

o de la cache L1 directamente. A diferencia de diseños anteriores la LDS no necesita los SIMDs para mover los datos a los registros vGPRs.

4.1.4. Unidad de memoria vectorial

El sistema de memoria de la arquitectura GCN es un sistema unificado que permite cachear tanto las lecturas como las escrituras. Además, soporta direccionamiento virtual y operaciones atómicas. La cache L1 de datos presenta un tamaño de 16 KB con 4 vías, un tamaño de bloque de 64B y política de remplazo LRU. Este primer nivel de cache sigue una política write-through, write-allocate con máscara de byte modificados.

Se mantiene la coherencia entre los datos de L1 y L2 a través de un modelo de consistencia relajado. Conceptualmente, la L1 es coherente a nivel de work-group siendo eventualmente coherente a nivel global. Esta política se implementa asegurando que las escrituras se han realizado L2 cuando el wavefront ha terminado. Las líneas de cache que han sido escritas en su totalidad permanecen en L1 tras la finalización del wavefront, mientras que las que han sido escritas parcialmente son invalidadas. Por otro lado, las operaciones de lectura disponen de un flag especial para indicar que el dato se traiga de L2 sin acceder a la L1, para mantener la coherencia con las modificaciones producidas en otros CUs.

Cuando una unidad SIMD emite un acceso de lectura o escritura se calculan todas las direcciones emitidas por el subwavefront y se combinan (*coalesce*) en uno o más accesos. En el mejor de los casos los 16 accesos que componen un subwavefront sólo necesitarán acceder a 1 bloque de la cache. Esta técnica

permite reducir radicalmente la latencia de lectura y el tráfico a L2 en las escrituras.

4.2. Resumen

En este capítulo se ha detallado la arquitectura del núcleo más reciente de AMD para GPUs, GCN, donde se han visto sus unidades principales (figura 4.1), incluyendo una breve síntesis del sistema de memoria. Como el sistema de memoria de las tarjetas gráficas es muy diferente al de las CPUs convencionales, se va a dedicar el siguiente capitulo a presentar en profundidad su jerarquía y funcionamiento.

Capítulo 5

Sistema de memoria

Como ya se comentó en la introducción, este trabajo se va a centrar en el estudio del sistema de memoria, en concreto el sistema de memoria de las tarjetas gráficas de AMD Southern Islands. En las últimas décadas, el sistema de memoria se ha convertido en el principal cuello de botella, debido a que la potencia de cómputo crece mucho más rápido que la velocidad de la memoria. Por esta razón la industria se ve obligada a invertir gran cantidad de recursos en desarrollar nuevas memorias capaces de suplir las necesidades de los procesadores actuales.

Si nos centramos en un entorno CPU, el problema principal de los accesos a memoria es su alta latencia. Esto es debido a que las CPU convencionales ejecutan principalmente código secuencial. Incluso si se hace uso de técnicas multinúcleo y multihilo, la cantidad de threads en ejecución no es suficiente para ocultar la latencia de memoria.

Por otro lado, en los sistemas GPU el código está programado para un procesador masivamente paralelo que es capaz de ejecutar miles de threads simultáneamente. Con esta cantidad de threads resulta mucho más fácil ocultar la latencia de acceso a memoria. En este contexto el impacto de la memoria en las prestaciones depende principalmente de su ancho de banda. Para poder suministrar grandes anchos de banda es necesario recurrir a tecnologías de memoria diferentes a las usadas en las CPU. Por ello, actualmente las GPUs requieren memorias GDDR5, las cuales proporcionan un ancho de banda muy superior a las memorias DDR3.

5.1. DDR3 vs GDDR5

Para comparar estas dos memorias se debe consultar los manuales de ambos estándares que han sido creados por JEDEC, una asociación dedicada

a la creación de estándares abiertos para la industria electrónica. En estos documentos se pueden encontrar muchas diferencias entre estos dos tipos de memorias, pero las principales características que distinguen a la tecnología GDDR5 de la DDR3 son las siguientes:

- La memoria GDDR5 dispone de una señal de reloj específica para la interfaz de datos. Esta señal de reloj se envía desde el controlador de memoria y funciona al doble de frecuencia que el reloj normal. Como se trata de una interfaz DDR (Double Data Rate), los datos pueden ser transferidos en cada flanco de subida o bajada del reloj de datos. La memoria GDDR5 puede alcanzar tasas de transferencia por pin de hasta 6400Mbps.
- La GDDR5 tiene una latencia de CAS que va desde los 5 hasta los 36 ciclos, dependiendo de la configuración; mientras que en la memoria DDR3 la latencia de CAS oscila entre los 5 y los 16 ciclos.
- La tensión de alimentación es de alrededor de 1V para la GDDR5, mientras que para la DDR3 varía entre 1.25V y 1.65V.

Debido a estas características, GDDR5 es más apropiada para ser usada como memoria principal de las GPUs, cuyas prestaciones dependen principalmente del ancho de banda de memoria. Por otra parte, el incremento de la latencia de GDDR5 en comparación a DDR3, hace esta última más adecuada para ser usada como memoria principal de las CPUs.

5.2. Funcionamiento del protocolo de acceso a memoria en las GPUs Southern Islands

Sobre los sistemas de memoria de las GPUs no hay apenas documentación publicada por los fabricantes, debido a que estos desean mantener sus diseños en secreto para dificultar la competencia. Esto conlleva un reto añadido a la investigación en los sistemas de memoria de las tarjetas gráficas existentes en el mercado. En particular, para sus tarjetas Southern Islands, AMD proporciona algo más de información que la existente para otras GPUs. Sin embargo, la información publicada sobre la interconexión de los módulos de memoria las tarjetas Southern Islands sigue siendo muy escueta. Esta sección resume parte de la información publicada para explicar cómo funciona el sistema de memoria en estas tarjetas.

El sistema de memoria de las GPUs Southern Islands consta de 3 niveles: L1, L2 y memoria principal (ver figura 5.1). El nivel L1 está formado por

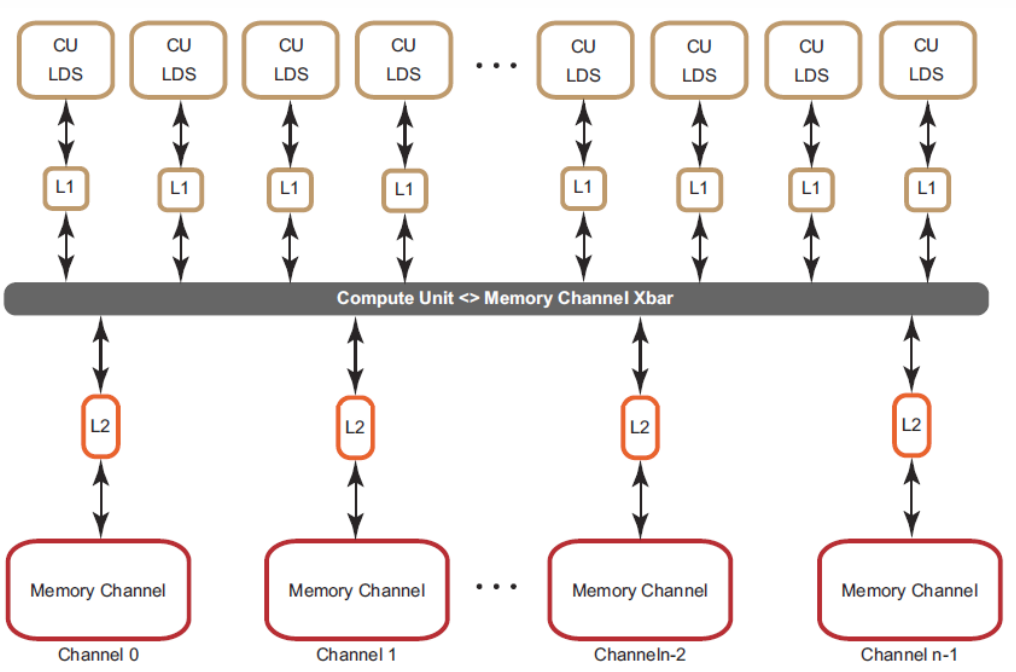


Figura 5.1: Jerarquía de memoria de las GPUs Southern-Islands.

módulos de cache de datos, en los cuales se puede leer y escribir; y módulos de constantes, que sólo son de lectura. Los módulos de datos son privados a cada CU y no coherentes entre sí, mientras que los de constantes son compartidos por cada 4 CUs. Este nivel sigue una política de escritura write-through.

El nivel L2 se comporta como una cache distribuida. Este nivel sigue una política de escritura write-back. Además, cada una de las líneas de cache cuenta con una máscara para indicar qué bytes de la línea han sido modificados. El nivel L2 está físicamente particionado en módulos, los cuales están conectados a un controlador de memoria de doble canal que enlaza con la memoria principal. Así, el número de módulos en L2 depende de los módulos de memoria con los que se equipa la tarjeta gráfica. La conexión con el nivel L1 se realiza a través de un crossbar, sobre el cual no hay información disponible.

La memoria principal esta dividida en 4, 8, o 12 módulos de memoria GDDR5, dependiendo del modelo de tarjeta. Los modelos más potentes tienen 12 módulos de memoria GDDR5, cada uno con su propio canal de comunicación con el controlador de memoria. Para evitar conflictos a nivel de canal, la memoria principal tiene las direcciones entrelazadas en superbloques de 256 bytes, es decir, 4 bloques convencionales. Los canales de acceso a memoria se agrupan en 4 cuadrantes que se seleccionan con los bits 9 y 10

de la dirección de memoria a la que se quiere acceder. Dentro del cuadrante se tiene que seleccionar un canal, y para ello se usa el bit 8 de la dirección. Como la gama de tarjetas de la serie 79XX tiene 12 canales por los que acceder a memoria principal, y como el número de canales no es una potencia de 2, dividir las direcciones sería muy costoso desde el punto de vista del hardware. Para agilizar este proceso se ha añadido un segundo bit, que se calcula usando la dirección del banco y la fila, para seleccionar el canal.

5.2.1. Coherencia relajada

El sistema de memoria implementa lo que AMD denomina coherencia relajada. Este tipo de coherencia consiste en tener un nivel de memoria para mantener la coherencia a nivel global entre todos los CUs, que en este caso sera el nivel L2, y otro nivel (L1) que sólo mantiene la coherencia entre los work-items pertenecientes a un mismo work-group. La idea subyacente es que los work-items se comuniquen a través de L1 y, eventualmente, cuando un work-group finaliza, hagan visibles sus modificaciones a nivel global a través de L2.

Para la implementación de este tipo de coherencia, el formato de las instrucciones vectoriales de acceso a memoria dispone de 2 bits que permiten modificar el comportamiento de las caches. Estos bits son el GLC (GLobal Coherent) y el SLC (System Level Coherent). Cuando el bit SLC está activo fuerza el acceso a memoria principal. Por otro lado, el bit GLC presenta 3 comportamientos distintos según el tipo de instrucción:

- En las instrucciones vectoriales de lectura si el bit GLC es igual a 0 se permite el acceso a L1, mientras que si se encuentra activo se fuerza el acceso a L2.
- Los bloques modificados parcial o completamente por las instrucciones vectoriales de escritura con el bit GLC a 1 son invalidados en L1. En caso de que el bit GLC sea igual a 0, sólo se invalidan si la modificación es parcial. En caso de que el bit GLC sea igual a 0 y el bloque sea modificado completamente, se limpia la máscara de dirty del bloque y se permite que continúe en L1.
- En las operaciones atómicas el bit GLC decide si se debe devolver el resultado anterior a la operación (GLC igual a 1) o si por el contrario sólo se debe modificar el valor (GLC igual a 0). Por ejemplo, si una operación atómica con el bit GLC activo incrementa una variable A cuyo valor inicial es 1, el resultado será que el valor almacenado en memoria será 2, mientras que el valor leído por la CU será 1.

5.3. Resumen

En este capítulo se han visto las principales diferencias entre las memorias DDR3 y GDDR5, y también se ha explicado con detalle la arquitectura del sistema de memoria que se implementa en la familia de GPUs Southern Islands. Se han indicado la distribución de los módulos de cache para los distintos niveles de la jerarquía de memoria, así como el funcionamiento de de los bits, SLC y GLC, que influyen en el comportamiento de los accesos a estos módulos.

Capítulo 6

Entorno de Simulación

Para la realización de este trabajo se necesita modificar la arquitectura de la GPU, y más concretamente el sistema de memoria. Para ello, se ha utilizado un simulador detallado de la GPU. Nótese que en el área de investigación en arquitectura de computadores, el desarrollo de simuladores de GPU se encuentra en sus inicios. Actualmente, los dos simuladores más usados en este área son GPGPU-sim y Multi2Sim.

Dos de las principales ventajas de Multi2Sim sobre GPGPU-sim es que el primero es capaz de soportar cargas heterógeneas e integra un framework OpenCL más avanzado. Ambas ventajas hacen más adecuado a Multi2Sim para el desarrollo del presente trabajo y la futura tesis doctoral en la que este se enmarca.

6.1. Multi2Sim

Multi2Sim es un simulador detallado de GPUs y CPUs superescalares, multinúcleo y multihilo. Actualmente puede emular 7 arquitecturas distintas, entre ellas la incluida en las tarjetas AMD Southern Islands. Multi2Sim permite observar la evolución del procesador ciclo a ciclo durante todo el proceso de ejecución. Presenta un modelo de simulación de 4 fases: desensamblador, simulador funcional, simulador detallado, y herramienta visual. El modelo del sistema de Multi2Sim se puede dividir en 3 partes:

- **Procesador**. El procesador puede ser de cualquiera de las 7 arquitecturas soportadas. Se pueden combinar procesadores de distintas arquitecturas en la misma simulación. Todas las arquitecturas son muy parametrizables, lo que ofrece gran flexibilidad para modificar los modelos originales.

- **Jerarquía de memoria**. La jerarquía de memoria representa los módulos de memoria del sistema a simular organizados en los distintos niveles. La jerarquía modela tanto los módulos de cache como los de memoria principal. Se pueden configurar una gran gama de características de cada módulo, tales como su geometría, política de reemplazo, latencia de acceso, número de puertos, etc. Para organizar la jerarquía, los módulos se conectan a otros módulos en niveles superior e inferior de la jerarquía. La jerarquía de memoria soporta el protocolo de coherencia NMOESI, descrito más adelante.

- **Red**. La red es la encargada de conectar todos los nodos del sistema, los cuales se asocian a módulos y procesadores. La red modela la contención que se puedan producir debido al tráfico que circula por los enlaces y los buffers de entrada o salida en los nodos. Se pueden modelar diversos tipos de switches, buffers, y políticas de enrutamiento.

La simulación detallada se implementa ejecutando un bucle por cada uno de los pipelines simulados, representando cada iteración un ciclo de reloj en el procesador real. El simulador detallado recibe el flujo de instrucciones del simulador funcional, y cuando finaliza su simulación del *timing* de una instrucción envía una petición para que el simulador funcional emule su comportamiento. El simulador detallado hace un seguimiento de la instrucción por todos los componentes la arquitectura simulada. Durante este proceso la instrucción puede requerir el acceso a varios recursos, tanto internos del procesador como del sistema de memoria. El simulador detallado se encarga de simular la contención en el acceso a estos recursos, añadiendo al tiempo de ejecución de la instrucción las latencias correspondientes.

Multi2Sim incluye una implementación adaptada de las librerías OpenGL, OpenCL y CUDA, permitiendo la compilación y simulación de las aplicaciones que hacen uso de ellas. Esto permite obtener información sobre las llamadas a las funciones de estas librerias en tiempo de ejecución.

Multi2Sim 4.0 añade a sus múltiples características la capacidad de simular la arquitectura AMD Graphics Core Next (GCN) que implementan todas las tarjetas gráficas de AMD desde la serie HD 7000. Además, incluye el protocolo de coherencia NMOESI, diseñado para procesadores heterogéneos con núcleos GPU y CPU. La principal aportación de este protocolo es que añade un nuevo estado 'N' al protocolo MOESI para CPUs. Este estado 'N' (de Non-coherent) permite que las GPUs realicen escrituras no coherentes en la jerarquía de memoria.

6.1.1. MOESI

El protocolo MOESI es un protocolo de coherencia con 5 estados implementado en muchas CPUs recientes. Este protocolo añade el estado *Owned* (O) al protocolo MESI. Cuando se produce un fallo de lectura en una cache de L1, el bloque requerido se trae y se almacena con los estados 'S' o 'E', dependiendo de si está o no almacenado en alguna otra cache de L1, respectivamente. Cuando se produce un fallo de escritura, el bloque se almacena en la cache con estado 'M'. Nótese que esto implica una política de escrituras *write-allocate*). Si una cache de L1 sirve un bloque con estado 'M' a otra cache del mismo nivel por culpa de un fallo de lectura, su copia local pasa a tener el estado 'O'. Los bloques marcados con los estados 'M' y 'O' pueden leerse o escribirse localmente sin provocar cambios de estado adicionales en la copia local.

6.1.2. NMOESI

Como se comentó previamente, el protocolo NMOESI es una ampliación del protocolo MOESI, que mejora las prestaciones de este último cuando se aplica en el sistema de memoria de una GPU. NMOESI añade un estado no coherente 'N' para aligerar y proveer de más flexibilidad al acceso a memoria de las GPUs. En particular, el nuevo estado 'N' sólo puede ser generado por las escrituras no coherentes de la GPU. NMOESI ha sido implementado en Multi2Sim para poder dar soporte tanto a la simulación de GPUs independientes como a la simulación de procesadores heterogéneos que incorporan CPU y GPU en el mismo chip.

Cuando se emite una escritura no coherente, el bloque se trae y se le asigna el estado 'N'. Pueden existir múltiples copias del mismo bloque con estado 'N' en diferentes cache y es posible escribir en todas ellas sin provocar acciones de coherencia. Por tanto, es responsabilidad del programador asegurar que work-items distintos no afectan a la misma parte del bloque. Cuando se reemplaza un bloque con estado 'N', sólo la parte modificada se actualiza en el nivel inferior de la jerarquía. Para ello, se utiliza una máscara de bits (*dirty mask*) que indica que parte del bloque ha sido modificada.

6.2. Sistema base: GPU Southern Islands

La GPU que se ha escogido para los experimentos es una AMD Southern Islands. En concreto, se ha preparado una configuración para simular una tarjeta similar a la del 7970 de AMD. La GPU cuenta con 32 unidades de cómputo (Compute Unit o simplemente CU).

La jerarquía de memoria es exactamente igual al modelo físico, presentando una cache de datos en L1 por cada CU, y otra cache para constantes compartida cada 4 CUs. El tamaño de ambos tipos de cache es de 16 KB. Por otro lado, en L2 hay una única cache compartida por todos los CUs y organizada en 6 módulos, cada uno de ellos de 128 KB, los cuales se conectan con memoria principal con 2 canales de comunicación y a través de controladores de memoria de doble canal.

6.2.1. Implementación de la arquitectura GCN en el simulador

Multi2Sim modela el pipeline de la arquitectura GCN por medio de varios módulos que se corresponden con las unidades principales de esta arquitectura. Cada módulo esta dividido en las etapas por las que pasan las instrucciones durante el transcurso de su ejecución. También se modelan los buffers que almacenan las instrucciones entre etapas.

La latencia de cada etapa y el tamaño de los buffers son configurables. El tiempo de ejecución de una instrucción que atraviesa el pipeline sin contención será la suma de las latencias de cada etapa. En el caso de instrucciones de acceso a memoria, la latencia total depende de la latencia del sistema de memoria, que variará para cada petición. Las instrucciones de acceso a memoria tienen un flag para indicar cuándo ha terminado el acceso. En el caso de las instrucciones vectoriales de acceso a memoria, este flag se activa cuando terminan todos los accesos generados por la instrucción.

La figura 6.1 presenta el pipeline de la unidad de acceso a memoria vectorial. Se pueden diferenciar buffers, etapas y la interfaz con componentes externos tales como la memoria global y los bancos de registros.

6.3. Mejoras en el modelo de Multi2Sim

6.3.1. Miss status holding registers

Los registros MSHR (Miss Status Holding Registers) son comúnmente usados en las cache de las CPUs. Sin embargo, Multi2Sim no los implementa. Debido a esto, el factor principal que limita el número de accesos en vuelo a memoria desde la GPU es el buffer donde se almacenan las instrucciones mientras se está realizando el acceso a memoria. Este buffer corresponde con el *memory buffer* de la figura 6.1.

La capacidad de este buffer permite almacenar alrededor de 32 instrucciones vectoriales. Esto implica que cada CU puede realizar simultáneamente

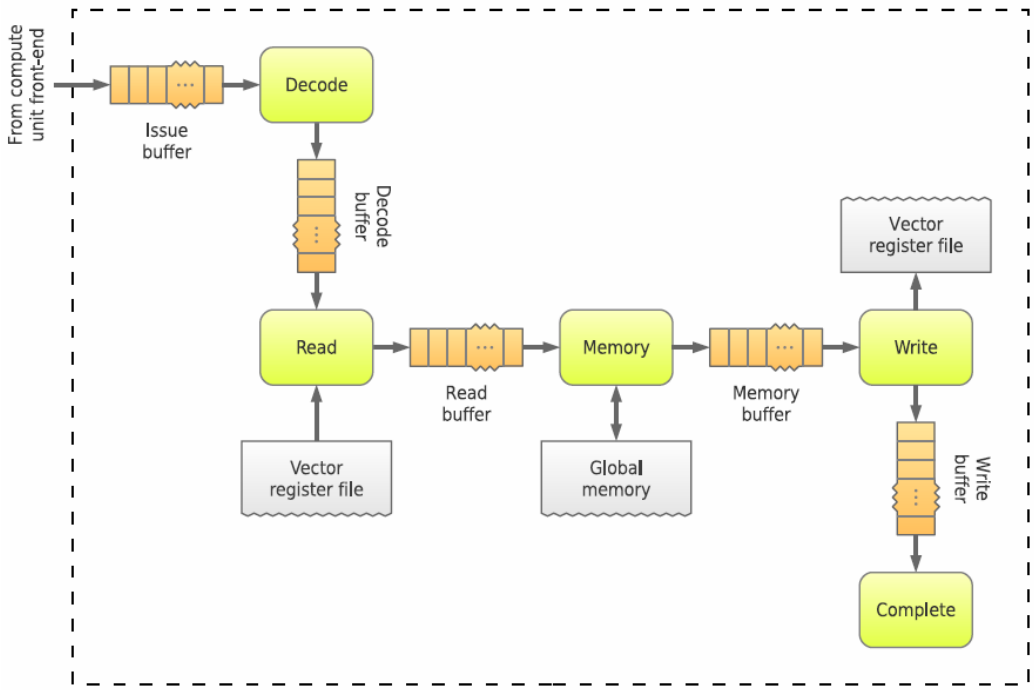


Figura 6.1: Implementación en Multi2Sim de la unidad vectorial de acceso a memoria de la arquitectura GCN

2048 accesos (32 instrucciones vectoriales x 64 threads/instrucción) a su cache de L1. Este número de accesos es excesivo para las cache de L1. Por tanto, se han modelado los registros MSHR para implementar un modelo más realista. Variando el número de estos registros, es posible limitar el número de peticiones que puede aceptar la cache.

6.3.2. Coalesce en L2

Una de las propiedades que permite a las GPU tener un alto rendimiento es que explotan el coalesce de los accesos a memoria. Las unidades de coalesce presentes habitualmente en el primer nivel de cache permiten juntar varios accesos, cuyas direcciones demandadas pertenezcan al mismo bloque, en una. En CPUs, esto puede presentar una leve mejora en el rendimiento; pero en las GPUs, el uso de esta técnica puede producir grandes ventajas, puesto que el número de peticiones a memoria es muy superior en comparación con el de las CPUs, y además, la metodología de programación favorece que los threads se organicen de forma similar a los datos. El coalesce se usa en el sistema de memoria del Multi2Sim, pero está limitado al nivel L1 de cache.

Antes de ver por qué podría ser interesante el coalesce en L2, vamos a ver los tipos de localidad que tenemos y cuáles de ellos se pueden beneficiar del coalesce o de la cache en el nivel L1. De esta forma vamos a acotar las situaciones en las que nos puede interesar tener el coalesce en L2.

Los kernels se pueden clasificar en dos tipos, dependiendo de si existe o no comunicación entre los distintos work-groups:

- Intra work-group: los kernels pertenecientes a este grupo no tienen comunicación entre los distintos work-groups del contexto. Por ello se puede ejecutar correctamente en un sistema de memoria sin protocolo de coherencia.

- Inter work-group: estos kernels tienen comunicación entre los work-groups. Esto significa que varios work-groups modifican la misma posición de memoria, pudiendo generar resultados erróneos si no se usan las barreras necesarias.

Dentro del grupo de kernels que tienen comunicación intra work-group podemos hacer una clasificación de los tipos de patrones de accesos que encontramos. Como podemos ver en el artículo de M. Martonosi [2] se pueden deducir 3 tipos de localidad a nivel de CU. Como este artículo está hecho desde el punto de vista de la arquitectura y nomenclatura de NVIDIA, vamos a definir la localidad desde el de la arquitectura Southern Islands:

- Localidad a nivel de wavefront: este tipo de localidad se da cuando, al ejecutar una load, todas las direcciones de los accesos generados, cada uno correspondiente a un work-item (thread), son direcciones consecutivas, es decir, pueden no caer todas en el mismo bloque porque el tamaño total de bytes pedidos sean mas grandes que un bloque de memoria. Por esta razón se tendrán que traer varios bloques consecutivos y sin datos no demandados entre ellos.

- Localidad a nivel de work-group: esta localidad se da cuando al ejecutar una load sobre wavefronts pertenecientes al mismo work-group, los datos de distintos wavefronts se encuentran mapeados en la misma línea de memoria. Este tipo de localidad se puede tratar de igual manera para las arquitecturas AMD como para las de NVIDIA.

- Localidad entre instrucciones: esta localidad se puede ver cuando los thread de un work-group ejecutan varias veces la misma instrucción y los datos de están en el mismo bloque. El caso típico será un bucle que va recorriendo un array progresivamente.

Una vez hemos definido los tipos de localidad que tenemos, hay que remarcar qué técnica beneficia a cada tipo de localidad.

La localidad a nivel de wavefront se va a ver completamente favorecida por el coalesce de L1, porque, juntado los accesos antes de que lleguen a la cache, nos estamos ahorrando congestiones en este módulo, y se pueden diseñar las caches de este nivel con un MSHR más pequeño. Por otra parte, para poder aprovechar bien este coalesce es necesario un trabajo extra por parte del programador, que tiene que hacer el kernel pensado en cómo se van a mapear estos datos a los threads que los van a utilizar, y aunque el entorno de programación de OpenCL lo facilite, en algunos casos puede seguir siendo un trabajo muy complicado.

Por otro lado, la localidad a nivel de work-group no puede ser explotada por el coalesce que hace la unidad de memoria vectorial, ya que ésta solo compara las direcciones generadas por la instrucción a nivel del wavefront. Debido a esto, este tipo de localidad puede beneficiarse de la cache L1, aunque hay que tener en consideración el caso en que todo el work-group ejecute la misma instrucción de acceso a memoria simultáneamente, ocupando los 4 wavefronts los SIMD de 1 core GCN. En este caso concreto, una segunda unidad de coalesce insertada en la propia L1 y dedicada a juntar accesos que busquen el mismo bloque pero procedan de distintos wavefronts, incluso aunque no fueran del mismo work-group, reduciría de forma notable el número de peticiones en vuelo en el sistema de memoria.

Por último, la localidad entre instrucciones no se va a poder beneficiar del uso de las unidades de coalesce, bien sea en la unidad de acceso a memoria vectorial o en la propia cache. Este tipo de localidad, sin embargo, se va a poder ver favorecido por el uso de caches.

En las caches L2, ambos grupos de kernels se pueden beneficiar en las lecturas, tanto de la unidad de coalesce como de la cache, pero son los kernels pertenecientes al grupo de los que tienen comunicación inter work-group los que tienen más posibilidad de compartir datos entre CUs.

Si se implementara una unidad de coalesce, algunas aplicaciones se podrían ver aceleradas al poder combinar peticiones de distintas CUs, reduciendo así la carga del sistema de memoria. Pero en un sistema de memoria con un protocolo de coherencia como el MOESI no se puede aplicar de forma eficiente el coalesce en L2, debido a que este protocolo bloquea las líneas de cache L1 antes de ir al nivel L2 para reservar la línea para cuando regrese la respuesta. Esto hace que si se bloquea una línea en L1 y se envía la petición a L2, ésta pueda hacer coalesce con otra y quedarse a la espera mientras la línea en L1 sigue bloqueada, algo que puede ralentizar el sistema de memoria debido a que si alguna petición tuviera que invalidar el bloque en L1, no podría, fallaría y haría un reintento, aplicándose a esta petición los retardos

de las fases por las que ha pasado, más un retardo hasta volver a intentar hacer el acceso.

6.3.3. Implementación de sistemas de memoria Southern Islands

Como el Multi2Sim solo implementa 1 protocolo de memoria y este es muy distinto al implementado en la Arquitectura de las GPU Southern Islands se ha decidido implementar dicho protocolo para poder compararlo con los demás.

A diferencia del protocolo NMOESI que es bastante complejo de implementar debido a que posee diferentes tipos de mensajes para poder dar soporte a la coherencia, el protocolo SI, como llamaremos a partir de de ahora el protocolo de memoria del southern islands, solo consta de 2 tipos de mensajes: de escritura (STORE) y de lectura (LOAD).

Ambos tipos de mensaje acceden al sistemas de memoria por el nivel L1, donde deben ocupar una entrada libre en el MSHR del modulo de L1 asociado a la CU para poder acceder a dicho modulo. Una vez se ha accedido al modulo se buscará el bloque y se bloqueara su entrada en la cache. En caso de que este no esté en la cache se le asignara una linea, invalidando si hace falta otro bloque siguiendo el orden LRU, a partir de este punto el comportamiento de los dos tipos de mensajes es distinto:

- Mensajes de lectura (LOAD): Dependiendo del bit GLC se permite hacer hit o no en el nivel L1. Así en caso de fallo, o hit con el bit GLC a 1, se mandara otra petición de lectura hacia el nivel L2. En este nivel se repite el procedimiento, pero sin tener en cuenta el bit GLC, es decir, se coger una entrada disponible en el MSHR, se busca el bloque y en caso de hit se devolverá la respuesta a L1. En caso de fallo se generara otra petición hacia memoria principal, la cual no tiene MSHR. Cuando de devuelve una petición desde un nivel superior se coloca el bloque en la linea de cache asignada y que se ha mantenido bloqueada para evitar que otra petición la ocupara mientras se estaba procesando la petición actual. Para finalizar la petición se libera la entrada del MSHR, se desbloquea la linea de cache accedida y se notifica a la CU que se ha finalizado dicho acceso, así cuando todo los accesos producidos por la misma instrucción finalicen está podrá seguir siendo ejecutada.

- Mensajes de escritura (STORE): Las escrituras se realizan simultáneamente en los niveles L1 y L2 del sistema de memoria. Cuando ya se tiene la linea en L1 bloqueada se manda otra petición de escritura al

nivel L2. Si el bit GLC es igual a 1 o encaso de escritura parcial del bloque de datos y el bit GLC valga 0, entonces se invalidará el bloque si esta ya se encontraba contenido en el modulo de cache L1 asociado a la CU. Una vez realizado esto la L1 uno dará por finalizado el acceso y liberara la linea de la cache. Mientras tanto en el nivel L2 escribirán los datos del bloque modificado en la cache de dicho nivel, invalidando si es necesario otro bloque, y marcando los byte modificados en la mascara de bytes modificados o dirty mask.

Las invalidaciones en L1 no tiene ninguna consecuencia, simplemente se limpia la linea de la cache y ya esta. Sin embargo, cuando se invalida un bloque en L2 se genera una petición de escritura (write-back) hacia memoria principal siempre y cuando dirty mask del bloque no sea 0. Hay que remarcar que en este protocolo las invalidaciones no añaden latencia a la petición principal, puesto que esta no tiene que esperarse a que se complete la copia del bloque invalidado y ambas operaciones puedes transcurrir en paralelo.

6.4. Cargas de GPU

Los experimentos han sido realizados utilizando algunos de los benchmark suministrados en el SDK de AMD, que ha adaptado el equipo de Multi2Sim para su simulador, compilándolo con su propia librería de OpenCL. Estos programas realizan operaciones comúnmente usadas en entornos de programación.

Estos benchmarks están compuestos de un programa escrito en C o C++ que es el programa que se ejecuta en el entorno host, el cual prepara el contexto OpenCL para ser lanzado sobre los dispositivos disponibles, y un archivo con extensión .bin que es el kernel precompilado con la librería OpenCL que implementa el Multi2Sim, que se lanzará sobre la unidad OpenCL seleccionada.

Capítulo 7

Resultados experimentales

En este capítulo se presentan los resultados experimentales que se han realizado para estudiar el comportamiento de los distintos protocolos estudiados: MOESI, NMOESI y SI, que se han visto en el capítulo 6.

El capítulo primero presenta las métricas de prestaciones estudiadas. Posteriormente, se presentan y analizan los resultados de los distintos estudios realizados.

7.1. Métricas sobre el comportamiento del sistema de memoria

En el estudio del comportamiento analiza la evolución temporal de distintas métricas de prestaciones relacionadas con los sistemas de memoria de las GPUs. A continuación se describen las principales métricas estudiadas.

Hit ratio. Esta magnitud indica el rendimiento que tiene cada módulo cache, medido en porcentaje de aciertos de los accesos realizados a dicho módulo. Debido a que el número de memorias cache es muy elevado y el comportamiento entre las caches del mismo nivel es bastante homogéneo, los resultados se estudian considerando las caches del mismo nivel. De esta forma, solo se analizan dos hit ratios, uno para las caches L1 y otro para las caches L2.

MPKI. El impacto del hit ratio sobre las prestaciones depende mucho del porcentaje de instrucciones de acceso a memoria que tenga la aplicación. Por ejemplo, un hit ratio bajo apenas influirá en las prestaciones de aplicación de calculo intensivo con muy pocos accesos a memoria. La métrica MPKI (siglas del término inglés Misses Per Kilo Instructions) cuantifica los fallos de cache cada 1000 instrucciones ejecutadas, y ayuda a comprender mejor el comportamiento del sistema de memoria, completando la información que se

pueda extraer del hit ratio. Al igual que con el hit ratio esta magnitud se calcula para cada nivel de cache. El MPKI cuantifica la cantidad de accesos que fallan al buscar el bloque en cada nivel, es decir, indica la cantidad de accesos que se generan hacia el nivel superior.

Coalesce. Como se ha descrito previamente la unidad de coalesce es una de las principales ventajas que proporcionan las GPUs, sin embargo su efectividad varía en función de la aplicación e incluso dentro de las distintas fases de ejecución de una misma aplicación. Esta métrica cuantifica la cantidad de accesos que se combinan en un único acceso efectivo para estudiar la eficacia de dicha técnica, y estimar si la unidad coalesce esta siendo efectiva o no. Esta métrica se estudiará para los distintos niveles de cache

Latencia de acceso a memoria. Este retardo comprende el tiempo desde que un acceso a memoria es emitido desde la CU hasta que finaliza. Es importante notar que una instrucción de acceso a memoria vectorial genera múltiples accesos independientes. En principio se podría considerar la latencia media de acceso a memoria, cuantificada como la media aritmética de todos los accesos, de manera análoga a como se estudia en los procesadores convencionales para estudiar las prestaciones del sistema de memoria. Sin embargo, esto nos llevaría a conclusiones erróneas en una GPU. Efectivamente, debido a las características del cómputo vectorial, la GPU o más concretamente la CU, bloquea las instrucciones posteriores a una load vectorial hasta que no finaliza el último acceso a memoria generado por ésta. Por tanto una opción más adecuada para estudiar las prestaciones del sistema de memoria en una GPU es considerar la latencia de una instrucción vectorial como una latencia única que finaliza en el momento en que termina su último acceso generado. Esta magnitud varia según la carga del sistema, el hit ratio, el MPKI o el protocolo de coherencia utilizado.

Entradas ocupadas. El número de entradas ocupadas hace referencia al grado de utilización del MSHR. Cuando el MSHR tiene todas sus entradas ocupadas, bloquea a todos los accesos que intenten acceder al módulo de cache asociado con dicho MSHR. Se puede deducir que una cantidad pequeña de entradas en el MSHR podría limitar seriamente las prestaciones potenciales de dicha aplicación. Esta métrica nos proporciona información sobre la carga que la aplicación estudiada induce en el sistema de memoria.

OPC. La métrica instrucciones por ciclo o IPC se ha utilizado clásicamente en la evaluación de prestaciones de procesadores superescalares, y nos indica el número medio de instrucciones ejecutadas por ciclo. Sin embargo, se deben hacer ciertas consideraciones cuando se utiliza en una GPU. En una GPU se utilizan instrucciones vectoriales e instrucciones escalares, ambas realizando un trabajo dispar. Mientras una instrucción vectorial puede lanzar decenas de operaciones (hasta 64 en la máquina estudiada) en un mis-

ma instrucción máquina, una instrucción escalar solo puede lanzar una. En consecuencia, mezclar ambos tipos de instrucciones en una única métrica no nos proporcionaría información alguna sobre la eficiencia de la GPU. Más aún, las instrucciones vectoriales pueden variar mucho su eficiencia, mientras que algunas lanzar unas pocas operaciones otras lanzar varias decenas. Debido a divergencias en el código no siempre se procesan 64 work-item u operaciones por instrucción. En consecuencia, es necesario la utilización de métricas auxiliares. En este trabajo se propone la utilización de la métrica OPC definida como operaciones por ciclo. Esta métrica considera, para cada instrucción vectorial, el número de operaciones distintas en la instrucción.

7.2. Análisis del impacto del tamaño del MSHR en las prestaciones

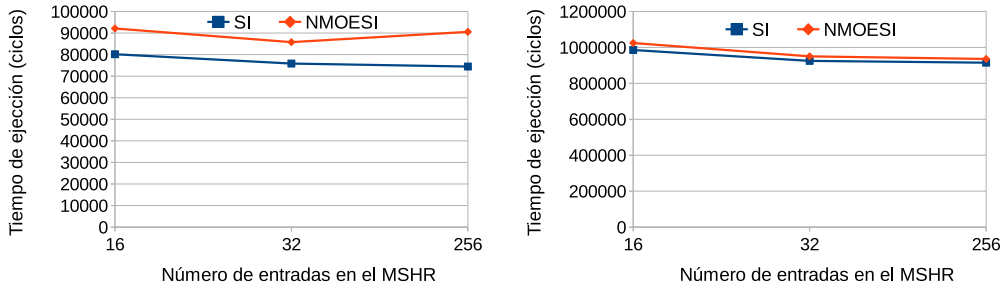
En los experimentos mostrados en esta sección se varía el tamaño del MSHR (cuantificado en número de entradas) de la cache L1 para los protocolos NMOESI y SI, y se estudia su impacto en el tiempo de ejecución. Se han usado 3 tamaños distintos: 32, 64 y 256 entradas, coincidiendo este último valor con el número de líneas de dicha cache. Este experimento permite estudiar el comportamiento de cada protocolo a medida que se permite un mayor número de peticiones de memoria en vuelo, esto es, se permite que se genere un mayor tráfico de coherencia, puesto el tamaño del MSHR acota el número de peticiones en vuelo.

A continuación se presentan los distintos estudios.

7.2.1. Tiempo de ejecución

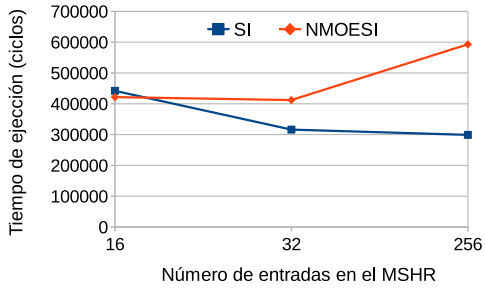
Tras el análisis del comportamiento individual de los distintos benchmarks para ambos protocolos, se han identificado cuatro patrones de comportamiento distintos. A modo de ejemplo, la figura 7.1 ilustra un ejemplo de cada tipo. A continuación se presentan los patrones y se describen las principales características de cada patrón.

- **SI siempre mejor**. Este patrón incluye los benchmarks en los cuales el protocolo SI ofrece mejores prestaciones independientemente del tamaño del MSHR. Es el caso del benchmark `blackscholes` que se presenta en la figura 7.1a. En la gráfica se aprecia que el protocolo SI reduce siempre el tiempo de ejecución respecto al NMOESI para los distintos valores del tamaño. Esta diferencia de prestaciones es siempre superior al 10 %.

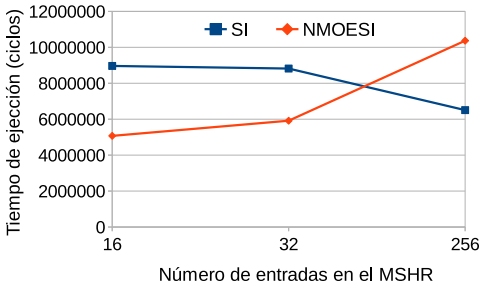


(a) blackscholes

(b) reduction



(c) mersennetwister



(d) matrixtranspose

Figura 7.1: Tiempo de ejecución los protocolos NMOESI y SI variando el tamaño del MSHR

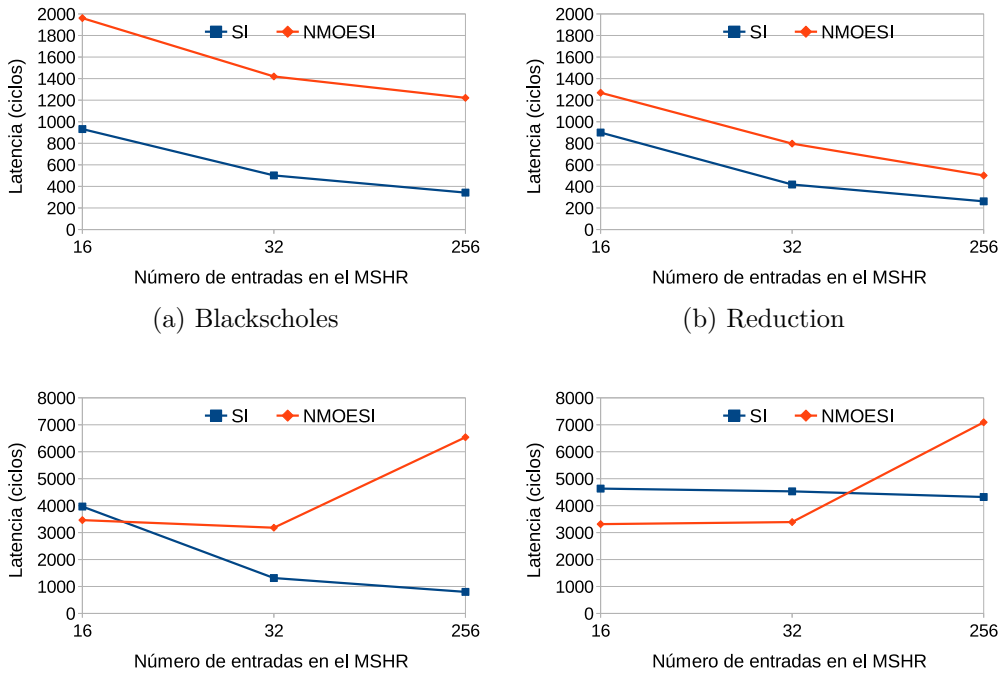
- **Comportamiento similar**. Este patrón incluye aquellos benchmarks en los que ambos protocolos presentan prestaciones muy similares para los distintos valores de MSHR. Como ejemplo de benchmark mostrando este comportamiento se encuentra el benchmark `reduction` que se presenta en la figura 7.1b

- **SI creciente**. En algunos benchmarks, ambos protocolos presentan prestaciones similares para un valor pequeño del MSHR, pero la diferencia aumenta en beneficio del protocolo SI a medida que se incrementa el tamaño del MSHR. Es el caso, entre otros, del benchmark `mersennetwister` que se presenta en la figura 7.1c

- **NMOESI mejor para pocas entradas**. Este patrón incluye los benchmarks que con un MSHR pequeño presentan mejor comportamiento para el protocolo NMOESI pero cuando cuando se aumenta el tamaño del MSHR esto se invierte siendo mejor el protocolo SI. A modo de ejemplo, se presenta el comportamiento del benchmark

`matrixtranspose` en la figura 7.1d.

Para entender el porqué de este comportamiento se han analizado distintas métricas.



(a) Blackscholes (b) Reduction

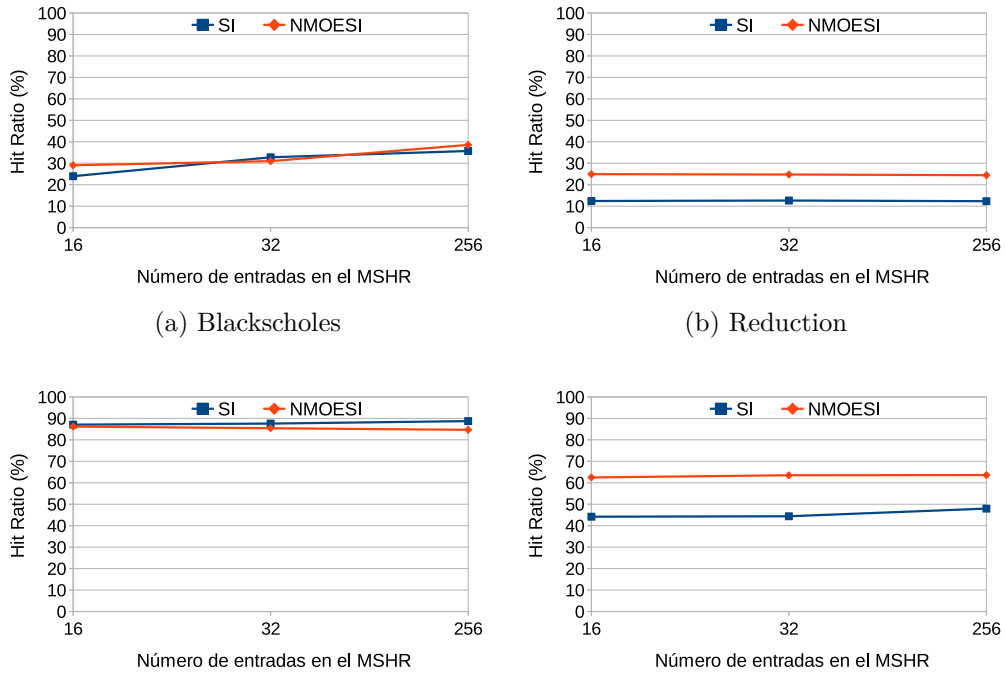
(c) Mersenne Twister (d) Matrix Transpose

Figura 7.2: Latencias para los protocolos NMOESI y SI, para distintos tamaños de MSHR.

7.2.2. Latencia de Memoria

La métrica que mejor explica el comportamiento es la "latencia de memoria" presentada en la figura 7.2 para los cuatro benchmarks estudiados anteriormente. Sin embargo, hay que tener en cuenta ciertas consideraciones importantes para entender la relación entre ambas figuras.

Para que la latencia afecte a las prestaciones debe superar el umbral de latencia que consigue ocultar la GPU al multiplexar en el tiempo la ejecución de los work-groups. Por ejemplo, en el benchmark `reduction` que se presenta en las gráfica 7.2b muestra que con un tamaño de 256 del MSHR se reduce la latencia más de la mitad respecto a 16 entradas para ambos protocolos; sin embargo, el tiempo ejecución de ambos protocolos apenas varía. Además,



(a) Blackscholes

(b) Reduction

(c) Mersenne Twister

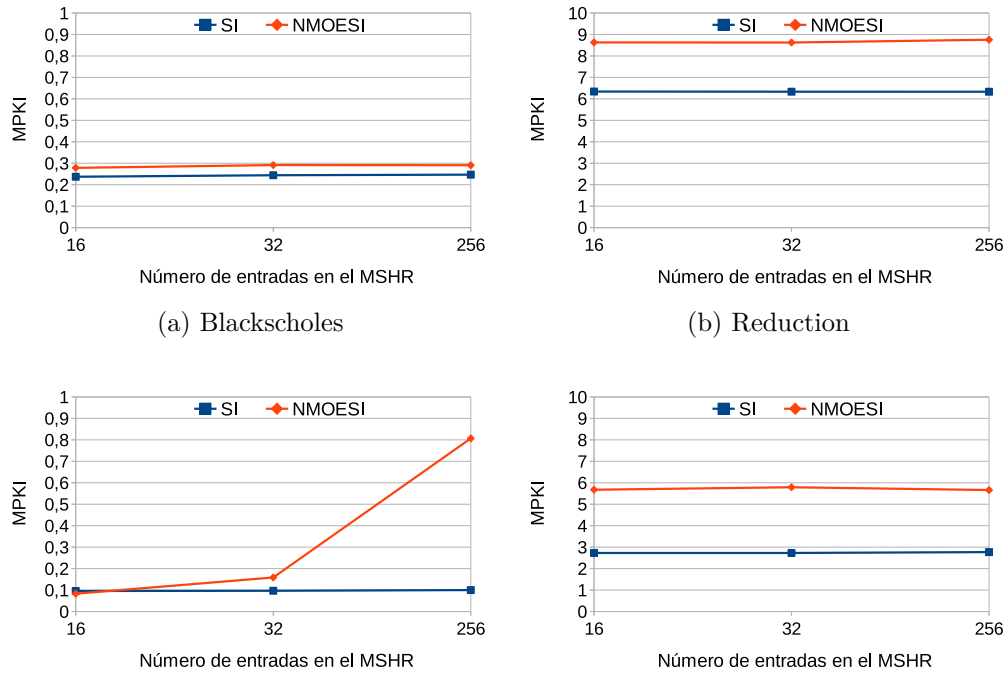
(d) Matrix Transpose

Figura 7.3: Hit Ratios para los protocolos NMOESI y SI, para distintos tamaños de MSHR.

también se aprecia que el protocolo SI consigue una reducción de latencia superior al 40 % respecto al NMOESI pero esta reducción no repercute positivamente en el tiempo de ejecución. En resumen, los valores de latencia mostrados oscilan entre 220 ciclos y 6× este valor, sin embargo este amplio rango muestra relación alguna en el tiempo de ejecución debido a que la GPU es capaz de ocultar estas latencias durante la ejecución de los work-groups.

Un efecto similar ocurre en el benchmark `blackscholes` mostrado en la gráfica 7.2a. En este caso se observa que la diferencia de latencia decrece en ambos protocolos notablemente a medida que aumenta el tamaño del MSR, sin embargo el tiempo de ejecución oscila muy poco para los distintos valores del MSHR. Por otra parte, el protocolo SI reduce la latencia a aproximadamente la mitad en para un MSHR de 16 y en un factor de 6 para un MSHR de 256 entradas. Sin embargo, el impacto sobre el tiempo de ejecución es siempre más o menos el mismo. Esto se debe a que la diferencia de latencia, cuantificada en valores absolutos, entre ambos protocolos es similar para los distintos tamaños de MSHR.

Las curvas de las latencias de `mersennetwister` y `matrixtranspose` y



(a) Blackscholes

(b) Reduction

(c) Mersenne Twister

(d) Matrix Transpose

Figura 7.4: MPKI para los protocolos NMOESI y SI variando el tamaño del MSHR.

mostradas en las figuras 7.2c y 7.2d, respectivamente, presentan una forma muy similar a las mostradas en las figuras de prestaciones. Las líneas se cruzan tanto para la gráfica del tiempo de ejecución como para la de latencia. Esto indica que la latencia de memoria puede ser un buen indicador del comportamiento que van a tener este tipo de aplicaciones. La razón principal es que las latencias de memoria de estos benchmarks así como las diferencias de éstas en valores absolutos, son en general, mucho mayores que en los benchmarks anteriores.

7.2.3. Tasa de aciertos en la cache y MPKI

La tasa de aciertos y el MPKI han sido métricas típicamente utilizadas en los procesadores para evaluar las prestaciones.

La latencia de los accesos a memoria descrita, en principio, se ve afectada por la tasa de aciertos en la cache o Hit Ratio. En las figuras 7.3 se analiza el Hit Ratio que presentan las aplicaciones tomadas como ejemplo. Sin embargo, no se ve ningún tipo de relación entre las latencias vistas previamente y el Hit

Ratio presentado. De hecho, el Hit Ratio se comporta de manera similar para todos los tamaños de MSHR. Lo que significa que esta métrica es irrelevante para el análisis del tiempo de ejecución de cada aplicación.

En la figura 7.4 se presenta el MPKI de los distintos benchmarks. Los benchmarks `blackscholes`, `reduction` y `matrixtranspose` presentan un MPKI similar aunque se varíe el tamaño del MSHR para un mismo protocolo. En los últimos benchmarks citados el MPKI es mucho menor para el protocolo SI. Sin embargo, en el benchmark `mersennetwister` se puede apreciar una variación notable con el valor del MPKI conforme aumenta el tamaño del MSHR, que imita la variación de la latencia de dicho benchmark mostrada en la figura 7.2c. Esto significa que el MPKI puede ayudar a predecir la latencia de memoria en algunos benchmarks, aunque intervienen más factores, posiblemente relacionados con el paralelismo de acceso a memoria que impide que esta métrica sea adecuada para otros benchmarks.

7.2.4. Resumen

En general, el protocolo NMOESI se comporta peor con 256 entradas, y a medida que se va restringiendo el número de peticiones en vuelo en el sistema su comportamiento mejora. Hay que destacar que si se reduce el tamaño del MSHR llegará un punto donde las prestaciones vuelvan a empeorar debido a que se estarán filtrando en gran medida los accesos a memoria, lo que incrementará ostensiblemente la latencia. Por su parte, eL protocolo SI sigue un comportamiento más escalable, mejorando según incrementa el tamaño del MSHR.

Si se consideran valores absolutos, el protocolo NMOESI es el que ofrece las mejores prestaciones con un MSHR de 16 entradas para muchas aplicaciones. Sin embargo, en algunas aplicaciones como Blackscholes, éste protocolo ofrece peores prestaciones que el protocolo SI.

Las observaciones anteriores nos llevan a concluir que ninguno de los dos protocolos es necesariamente mejor que el otro sino que depende del tipo de aplicación y número de entradas. Por ejemplo, mientras que en el benchmark `blackscholes` lo mejor es utilizar el protocolo SI con 256 entradas en `matrixtranspose` el protocolo con mejores prestaciones es el protocolo NMOESI con 16 entradas.

Esta diferencia que presentan dichos protocolos es muy importante ya que el tamaño de las GPU está creciendo muy rápidamente. La última gama de tarjeta de AMD tiene un 35 % más de CUs que su gama anterior (southern-Islands vista en este trabajo), por lo que se requieren protocolo de acceso a memoria que sean escalables y toleren un alto número de peticiones en vuelo.

7.3. Prestaciones del protocolo SI

En este experimento se comparan las prestaciones de los protocolos SI y NMOESI, tomando como base el protocolo MOESI. Al igual que en la sección anterior se estudiarán tres tamaños de MSHR: 16, 32 y 256. Con los resultados obtenidos se analizará el speedup y la latencia de memoria para cada configuración, obteniendo de esta forma la configuración optima para cada benchmark.

En la figura 7.5 se pueden apreciar los speedups para cada benchmark con los distintos protocolos. Se observa cómo los protocolos no coherentes obtienen una ventaja sustancial en algunos benchmarks (BlackScholes, DwtHaar1D, QuasiRandomSequence, RadixSort, Reduction).

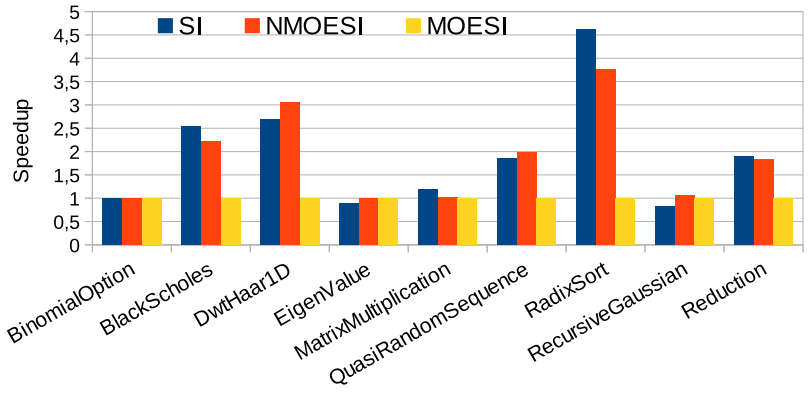
En BinomialOption, las prestaciones son iguales para los 3 protocolos. Esto se debe a que todas las configuraciones estudiadas son capaces de ocultar la latencia de los accesos a memoria.

En el caso del benchmark EigenValue, en el cual la latencia de acceso a memoria para los protocolo MOESI y NMOESI es muy baja (aproximadamente 3 ciclos de GPU) gracias al alto hit ratio que obtiene en la cache L1, que se sitúa entorno al 99 %. Sin embargo, cuando se ejecuta este mismo benchmark con el protocolo SI la latencia se incrementa hasta 75 ciclos, causando una penalización en el OPC.

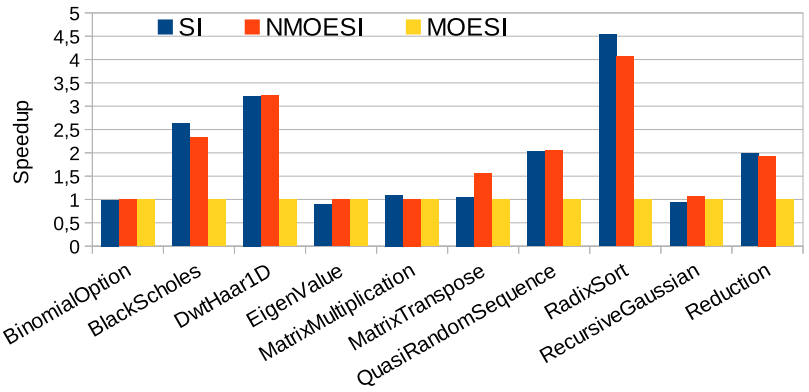
Los dos protocolos no coherentes presentan un comportamiento muy parecido, en cuanto a productividad throughput si se comparan con el protocolo MOESI, pero si comparamos el protocolo SI con el NMOESI se pueden apreciar ciertas diferencias. La variación del tamaño del MSHR no afecta igual a ambos protocolos, haciendo que en algunos casos se obtenga un rendimiento más alto con un MSHR más pequeño o viceversa. Esto plantea la posibilidad de ajustar el sistemas donde se va a lanzar una aplicación, al comportamiento de ésta. Este ajuste se puede llevar acabo de dos formas:

- **Configuración estática**. Para configurar la GPU de forma estática se requiere un análisis previo de la aplicación. Para este análisis se han utilizados 3 posibles valor del MSHR con 16, 32 y 256 entradas, y los 3 protocolos estudiados en este trabajo. Sobre estas configuraciones se han ejecutado todas las aplicaciones. En la figura 7.5 se pueden ver los resultados obtenidos en dichas pruebas. Basándose en estas gráficas se puede saber fácilmente qué protocolo da mejor resultado para cada tamaño de MSHR. Una vez se sabe la configuración más conveniente para cada benchmark se podría hacer que el sistema se reconfigurara para esta. Obteniendo así el máximo rendimiento en cada benchmark.

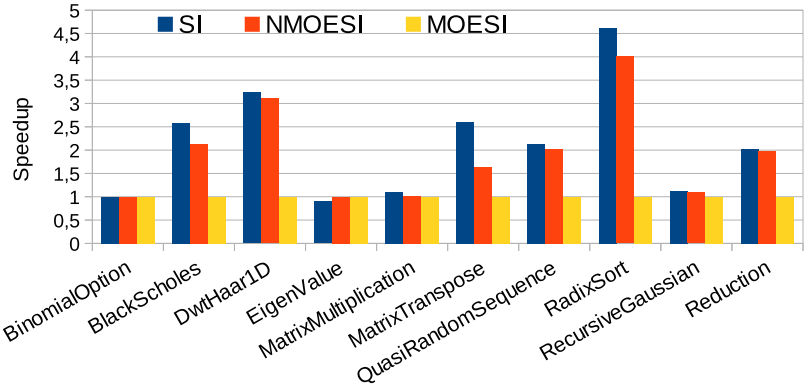
- **Configuración dinámica**. La configuración dinámica no requiere un



(a) speedup MSHR16



(b) speedup MSHR32



(c) speedup MSHR256

Figura 7.5: Speedup variando el MSHR respecto al protocolo MOESI

análisis previo, pero requiere implementar un sistema de muestreo que pueda recabar información de la aplicación que se está ejecutando. Mientras que la configuración estática se mantiene fija durante toda la ejecución de una aplicación dada, la configuración dinámica podría adaptar el hardware si detectara varios patrones de funcionamiento dentro de la misma aplicación. De esta forma se podría optimizar más aún el rendimiento de la GPU.

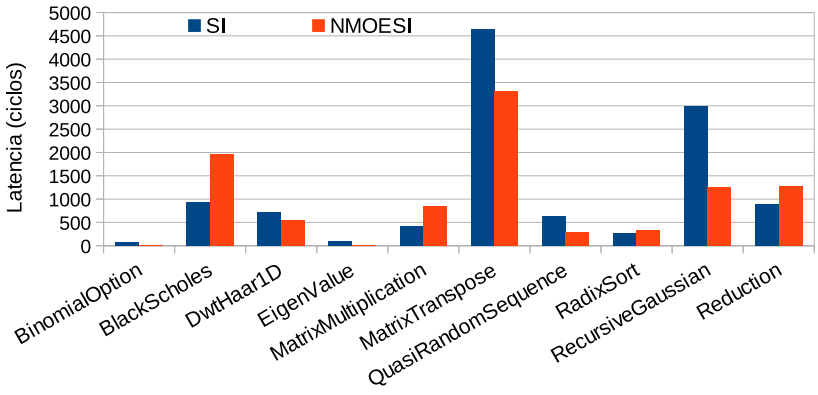
Como la latencia de acceso a memoria es un buen indicador de las prestaciones, también se han sacado los valores de las latencia para las configuraciones anteriores. En las gráficas 7.6a, 7.6b y 7.6c se puede ver como en general la latencia baja para casi todos los benchmarks según se va aumentando el tamaño de MSHR. En los únicos casos donde se produce un incremento de la latencia al aumentar el tamaño de MSHR es en el protocolo NMOESI con los benchmarks: MatrixTranspose y RecursiveGaussian.

En el caso de MatrixTranspose, el protocolo SI presenta una latencia casi constante para las 3 configuraciones, con una latencia entorno a 4500 ciclos. Sin embargo, los resultados para la misma aplicación con el protocolo NMOESI presentan un incremento notable de la latencia, siendo la mejor configuración para este protocolo el MSHR de 16 entradas que presenta una latencia de unos 3200 ciclos, llegando a una latencia de 7000 ciclos para la configuración con 256 entradas.

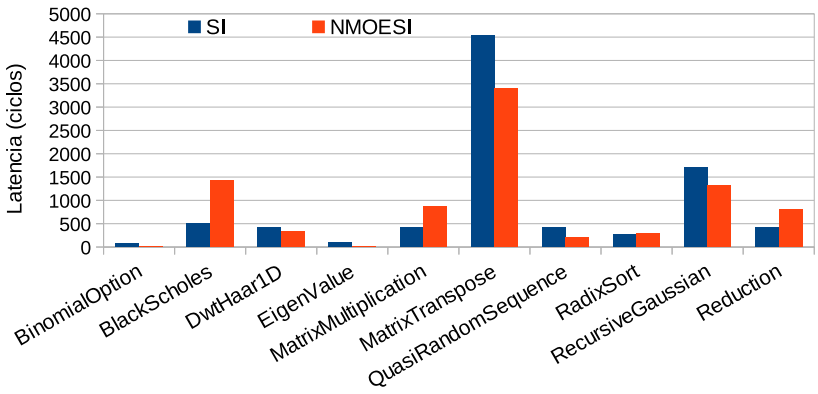
Con RecursiveGaussian, en el que el protocolo SI baja su latencia de 3000 a 1000 ciclos, mientras con el protocolo NMOESI el incremento de entradas en el MSHR hace que la latencia suba desde 1200 a 2500 ciclos. Este comportamiento que presenta la latencia de ambas aplicaciones hace que el NMOESI se comporte mejor en las configuraciones con poco número de entradas en el MSHR, pero cuando incrementamos el MSHR a 256 entradas el protocolo SI se ve bastante beneficiado.

En general, el protocolo SI empeora el rendimiento de las caches obteniendo un hit ratio más bajo, pero a la vez consigue reducir la latencia de acceso a memoria. Esto es posible debido a la ausencia de mensajes de coherencia, que hace que sea más rápido invalidar bloques en la jerarquía de cache. En ese sentido, el protocolo SI es más efectivo reduciendo la latencia que percibe la CU.

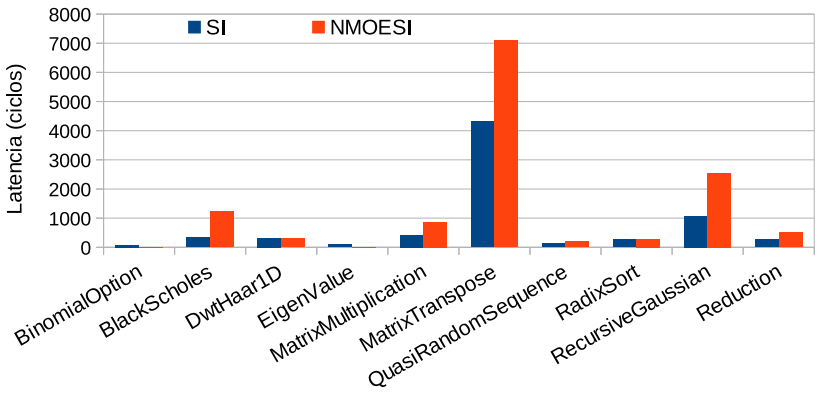
Por otro lado, algunos de los benchmarks presentan una tolerancia a la latencia muy alta. Esta tolerancia depende de cómo esté programado el kernel, siendo acotada por la cantidad de work-groups que pueda tener en ejecución cada CU. Así, queda en evidencia la importancia de la metodología de programación usada en la fase de desarrollo de los kernels. Para conseguir una tolerancia a la latencia alta no sólo basta con utilizar todas las CUs del



(a) MSHR 16

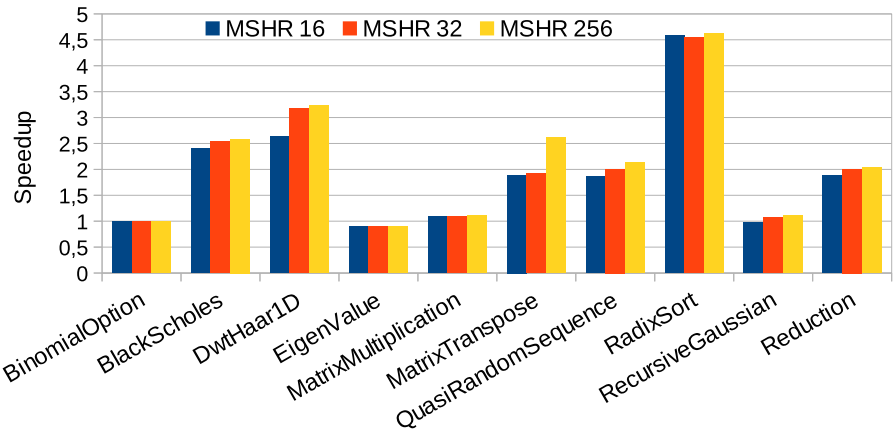


(b) MSHR 32

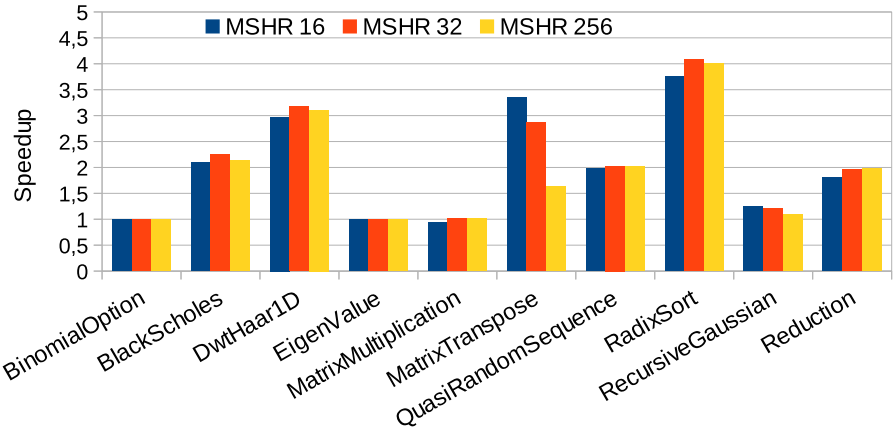


(c) MSHR 256

Figura 7.6: Latencias variando el MSHR para los protocolo SI y NMOESI



(a) speedup si



(b) speedup nmoesi

Figura 7.7: Speedup para cada protocolo, cogiendo como base el MOESI con MSHR 256

sistema, sino que además es necesario dividir el trabajo en suficientes work-groups para disponer de un alto número de wavefronts cuya multiplexación en el tiempo permita ocultar la latencia de memoria. De esta forma las CUs pueden tolerar latencias superiores, ya que tienen más wavefronts para multiplexar su ejecución. Esto ocurre porque cuanto más alto se consiga situar la latencia tolerada, las prestaciones se verán menos afectadas por el hardware donde se lancen.

Por último, se presenta en la figura 7.7 una comparación del speedup de cada protocolo para todas las combinaciones de MSHR, cogiendo como base el

protocolo MOESI con la configuración que obtiene las máximas prestaciones para dicho protocolo.

En la gráfica 7.7a se puede ver el speedup para todas las configuraciones de MSHR con el protocolo SI. En ella se puede ver que el SI obtiene mayor beneficio en todas las aplicaciones según se incrementa el tamaño del MSHR. Esto indica que para el protocolo SI lo mas favorable es tener un MSHR grande.

Sin embargo, la gráfica 7.7b, correspondiente al speedup del protocolo NMOESI frente al MOESI, muestra que los valores de MSHR óptimos para cada aplicación son distintos. En esta gráfica se encuentran benchmarks que se comportan igual para todas las configuraciones, y otros que presentan un speedup mejor para una configuración concreta. Entre estos últimos se pueden encontrar aplicaciones que funcionan mejor para cada una de las 3 configuraciones. De esta forma, para ejecutar aplicaciones en un sistema con el protocolo NMOESI de forma óptima, sería necesario poder variar el tamaño del MSHR, ya bien sea de forma dinámica o estática.

7.4. Resumen

En este capitulo se ha estudiado las prestaciones de los protocolos NMOESI y SI variando el numero de entradas en el mshr. Se han obtenido tiempos de ejecución para diversos benchmarks, así como latencias medias de memoria, Hit Ratios y MPKI.

Los resultados indican que las prestaciones de los protocolos están influenciadas por el tamaño de mshr, aunque en sentido distinto en función del benchmark y protocolo usado.

Finalmente se comprueba que métricas como el MPKI o el Hit Ratio, que en las arquitecturas de CPU son un buen referente para estimar las prestaciones, no proporcionan la suficiente información en ese sentido para las GPUs.

Capítulo 8

Conclusiones

En este trabajo final de máster han desarrollado los cuatro puntos especificados en los objetivos, que básicamente consistían en el estudio de los sistemas de memoria de la GPU con el fin de desarrollar protocolos de coherencia para sistemas heterogéneos. A diferencia de los protocolos que se utilizan en las CPUs, los protocolos que se utilicen en las GPUs de un sistema (heterogéneo o no) deben estar preparados para soportar el alto nivel de paralelismo que estas ofrecen, ya que pueden haber miles de peticiones en vuelo. En caso contrario, se pueden formar importantes cuellos de botellas en las prestaciones del sistema. A continuación se detalla el trabajo realizado en estos cuatro puntos.

Se ha implementado un protocolo reciente que se utilizan GPU actuales de AMD a partir de la información disponible publicada por AMD. Este protocolo se ha denominado SI, por la siglas de las tarjetas gráficas Southern-Islands que lo implementan. El protocolo SI implementa una coherencia relajada, lo cual hace que sea muy distinto a los demás protocolo visto en este trabajo. También se profundiza en la arquitectura de las GPUs actuales, presentando el core más reciente de AMD, denominado Graphics Core Next. Este core se encuentra implementado en todas las tarjetas gráficas de AMD desde la aparición de la Southern-Islands. A parte de ver las características del hardware, también se ha explicado el estándar OpenCL, el cual es el utilizado para programar los kernels que se lanzan sobre las GPUs.

Se ha realizado un estudio del comportamiento del protocolo SI y el protocolo académico NMOESI, que se encuentra implementado en el simulador Multi2Sim de amplio uso en la comunidad científica para la investigación en entornos GPU y CPU. Estos dos protocolos se han comparado con el protocolo MOESI implementado en la mayoría de CPUs actuales. El estudio se ha hecho para distintas cargas del SDK de AMD variando el número de entradas en el MSHR. Esta variable determina la cantidad máxima de peticiones que

se pueden tener en vuelo y, por lo tanto, tiene una relación directa con el tráfico de coherencia soportado por la jerarquía de memoria.

Se presenta un estudio del comportamiento de los tiempos de ejecución y analizado porqué varía dicho tiempo en una misma aplicación para los distintos protocolos variando el tamaño del MSHR. Para ello se ha estudiado la correlación del tiempo de ejecución con distintas métricas de prestaciones. En este estudio se muestra que la métrica que mejor explica el tiempo de ejecución en las aplicaciones GPGPU es la latencia de memoria. El MPKI también muestra cierta correlación pero en menor medida que en las cargas de CPU. Por su parte, el Hit Ratio no muestra relación alguna.

Los resultados muestran que ninguno de los dos protocolos estudiados es mejor que el otro sino que varía en función del tráfico que tenga el sistema de memoria, el protocolo utilizado y la aplicación.

Paralelamente a la escritura de este trabajo estamos preparando un paper con los principales resultados mostrados, y se prevé enviarlo a un congreso internacional en las próximas semanas.

Como trabajo futuro se prevé seguir con la línea marcada por el trabajo para la realización de la tesis doctoral. Como primer paso se pretende diseñar un protocolo adaptativo que ajuste el número máximo de peticiones en vuelo soportado a las características de la carga, ya bien sea de forma estática o dinámica en tiempo de ejecución. Posteriormente, se integrará el protocolo en arquitecturas heterogéneas para controlar la posible congestión del sistema de memoria debido a los accesos no coherentes.

Bibliografía

[1] Blake A. Hechtman and Daniel J. Sorin. *Exploring Memory Consistency for Massively-Threaded Throughput-Oriented Processors.*

[2] Wenhao Jia, Kelly A. Shaw, Margaret Martonosi. *Characterizing and Improving the Uses of Demand-Fetched Caches in GPUs.*

[3] Advanced Micro Devices. *GRAPHICS CORES NEXT (GCN) ARCHITECTURE whitepaper*, 2012.

[4] Shuai Mu, Yandong Deng, Yubei Chen, Huaiming Li, Jianming Pan, Wenjun Zhang, and Zhihau Wang. *Orchestrating Cache Managemet and Memory Scheduling for GPGPU Applications*

[5] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, David A. Wood. *Heterogeneous System Coherence for Integrated CPU-GPU Systems.*

[6] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, Tor M. Aamodt. *Cache Coherence for GPU Architectures.*

[7] Ahmad Lashgar, Amirali Baniasadi, Ahmad Khonsari. *Towards Green GPUs: Warp Size Impact Analysis.*

[8] Wenhao Jia, Kelly A. Shaw, Margaret Martonosi. *MRPB: Memory Request Prioritization for Massively Parallel Processors.*

[9] Jianlong Zhong, Bingsheng He. *Kernelet: High-Throughput GPU Kernel Executions with Dynamic Slicing and Scheduling*

[10] Feng Ji, Heshan Lin, Xiaosong Ma. *RSVM: a Region-based Software Virtual Memory for GPU.*

[11] Hanjin Chu, director, game , software and heterogeneous solutions. *AMD heterogeneous Uniform Memory Access.*

[12] Alameu Sankaranarayanan, Ehsan K. Ardestani, Jose Luis Briz, and Jose Renau. *An Energy GPGPU Memory Hierarchy with Tiny Incoherent Caches.*

[13] Shuai Mu, Yandong Deng, Yubeu Chen, Huaiming Li, Jianming Pan, Wenjun Zhang, and Zhihau Wang. *Orchestrating Cache Management and Memory Scheduling for GPGPU Applications.*

[14] George Kyriazis, AMD. *Heterogeneous System Architecture: A Technical Review.*

[15] Kyle Spafford, Jeremy S. Meredith, Seyong Lee, Dong Li, Pjilip C. Roth, Jeffrey S. Vetter. *The Tradeoffs of Fused Memory Hierarchies in Heterogeneous Computing Architectures.*

[16] Ahmad Lashgar, Amirali Baniasadi, Ahmad Khonsari *Towards Green GPUs: Warp Size Impact Analysis*

[17] N. Brunie. *Simultaneous Branch and warp Interweaving for Sustained GPU Performance*

[18] W. W. L. Fung, TorM. Aamodt. *Thread Block Compaction for Efficient SIMT Control Flow*

Índice de figuras