Memoria Transaccional Hardware en Memoria Local de GPU

Alejandro Villegas, Ángeles Navarro, Rafael Asenjo, Oscar Plata[1]

Resumen— **Los aceleradores gráficos (GPUs) se han convertido en procesadores de propósito general muy populares para el cómputo de aplicaciones que presentan un gran paralelismo de datos. Su modelo de ejecución SIMT (Single Instruction - Multiple Thread) y su jerarquía de memoria son piezas clave en la alta eficiencia de estas arquitecturas, que permiten el manejo de cientos o miles de hilos de ejecución. La jerarquía de memoria está dividida en dos espacios direccionables: Una memoria local, pequeña, rápida y visible por un subconjunto de los hilos en ejecución; y una memoria global, mayor, más lenta y visible por todos los hilos. Sin embargo, el modelo de programación SIMT no es eficiente cuando hay que sincronizar este desbordante número de hilos para garantizar exclusión mútua en una sección crítica. Utilizar atómicos para implementar cerrojos es problemático e ineficiente en este tipo de modelo de programación.**

La memoria transaccional (TM) ha sido propuesta como una alternativa más fiable y eficiente que los cerrojos para esta sincronización. Con TM, se permite el acceso especulativo a la sección crítica, registrando los accesos a memoria, deshaciendo los cambios de aquellos hilos que han tenido un conflicto y reiniciando su ejecución. En este trabajo presentamos una solución TM hardware que sincroniza aquellos hilos de ejecución que comparten la memoria local. En las pruebas realizadas, el uso de TM permite conseguir aceleraciones superiores a las soluciones basadas en cerrojos de grano grueso, asÃ como igualar a aquellas basadas en cerrojos de grano fino, pero con un menor esfuerzo de programación.

Palabras clave— **Arquitecturas GPU, Memoria Transaccional Hardware**

I. Introducción

Los procesadores gráficos (GPUs) han sido adoptados como aceleradores muy populares en aplicaciones que presentan un gran paralelismo de datos gracias a su modelo de ejecución *Single Instruction - Multiple Thread* (SIMT), su jerarquía de memoria y la disponibilidad de cientos o miles de núcleos de ejecución. Tecnologías como CUDA [1] y OpenCL [2] permiten el acceso a este hardware para el cómputo de propósito general. En este paper utilizamos la nomenclatura de OpenCL. Una GPU está compuesta de varios núcleos SIMT llamados *compute units* (CU). Los hilos de ejecución se denominan *work-items* y se agrupan en *work-groups*. Un programa a ejecutar se denomina *kernel*, y está compuesto por varios work-groups. En número de work-groups y work-items es definido por el programador. Un work-group es siempre planificado a una misma CU, y una CU puede ejecutar varios work-groups. Los recursos hardware de la CU son repartidos estáticamente entre todos los work-groups, propiciando cambios de contexto muy ligeros. Dentro de la CU, los work-items de un mismo work-group son agrupados en *wavefronts* de tamaño fijo. Dentro de la CU, el wavefront es la unidad planificable. Una CU posee dos espacios de memoria direccionables por los work-items: memoria global y memoria local. La memoria global es accedida por todos los work-items que se encuentran en ejecución. Esta memoria, de gran tamaño, tiene una gran latencia (en parte aliviada por una jerarquía de caches no coherentes) y se puede utilizar para comunicar work-items de work-groups planificados en distintas CUs, además de ser la que comunica la GPU con la CPU anfitrión. La memoria local tiene una latencia y tamaño menores. Existe un espacio de memoria local en cada una de las CU. Los work-items de un work-group planificado en una CU tienen acceso a esta memoria de forma compartida. Debido a su baja latencia, la memoria local es utilizada como *scratchpad* por los work-items de un mismo work-group. Work-items pertenecientes a diferentes work-groups planificados sobre la misma CU no comparten la memoria local de forma lógica (esto es, no pueden utilizarla para comunicarse entre ellos), pero sí de forma física. Además de estos dos espacios direccionables, cada work-item posee su propio espacio de memoria privado, típicamente mapeado en registros.

En general, las aplicaciones paralelas con múltiples hilos de ejecución deben utilizar mecanismo explícitos para la sincronización. Esta sincronización puede deberse a la necesidad de establecer secciones críticas en las que la exclusión mútua esté garantizada. Sin embargo, en el modelo de ejecución SIMT esto supone un reto mayor que en otras arquitecturas. Una solución típica es serializar la ejecución. En este caso, sólo un hilo ejecuta la sección crítica de forma secuencial, limitando el paralelismo de la aplicación. Otra solución consiste en delegar la ejecución de la sección crítica a la CPU anfitrión. Sin embargo, si los datos protegidos por la sección crítica se encuentran en memoria local, habría que copiarlos a memoria global, y luego al espacio direccionable por la CPU, lo que requiere una gran cantidad de ciclos de reloj. Otra posible solución consiste en implementar un mecanismo de sincronización basado en cerrojos utilizando operaciones atómicas. Implementar cerrojos de grano grueso es una solución fácilmente adoptable por los programadores. Sin embargo, el acceso a la sección crítica se hace en serie, perjudicando el rendimiento. Los cerrojos de grano fino, a priori, pueden ser una solución más eficiente. Sin embargo, es más difícil de implementar y propenso a *deadlocks* y *livelocks*.

La memoria transaccional (TM) [3], [4] se ha propuesto como una alternativa prometedora al uso de

[1]Universidad de Málaga, Andalucía Tech, Dept. of Computer Architecture, Spain. e-mail:{avillegas, angeles, asenjo, oscar}@ac.uma.es

cerrojos. El concepto de transacción se propone como complemento a la sección crítica. Al igual que una sección crítica, una transacción ejecutada por un hilo de ejecución debe garantizar exclusión mútua. Sin embargo, se permite el acceso concurrente a la transacción de forma especulativa por parte de todos los hilos de ejecución. Para garantizar exclusión mútua, todos los accesos a memoria deben ser monitorizados en busca de conflictos con otros hilos de ejecución. Las transacciones que encuentran un conflicto deben deshacer los posibles cambios especulativos en memoria y reiniciar su ejecución. Las que finalizan sin conflicto, pueden hacer definitivos sus cambios en memoria y continuar la ejecución. Existen diversas soluciones TM en arquitecturas CPU [11]. En las arquitecturas GPU están empezando a aparecer los primeros trabajos de TM, tanto software [5], [6], [7] como hardware [8], [9].

Dado que las CPUs actuales comienzan a tener soporte TM por hardware, extender este soporte a GPUs es un campo de trabajo importante y con gran impacto. Los actuales trabajos de investigacion en TM por hardware para arquitecturas GPU [8], [9] únicamente consideran el espacio de memoria global. Además, estas propuestas requieren cambios significativos en el hardware y organización de la GPU, motivos por los que los fabricantes pueden estar menos motivados para su implantación. Además, el espacio de memoria local no ha sido tenido en consideración. Este espacio de memoria es importante para los programadores, puesto que es utilizado para mejorar de forma significativa el rendimiento de sus aplicaciones. Por estos motivos, queremos dar un soporte TM hardware en GPU que sea ligero, eficiente, y que cubra todos los espacios de memoria direccionables. Asímismo, nuestra propuesta es implementada de forma incremental: en una primera etapa damos soporte TM hardware a los work-items de un mismo work-group que comparten la memoria local, mientras que en una segunda etapa extendemos esta implementación para dar soporte a todos los work-items de la GPU que se comunican mediante la memoria global.

Este artículo presenta una propuesta de TM hardware ligero y efectivo que permite definir transacciones que operan sobre la memoria local compartida por los work-items de un mismo work-group en una arquitectura GPU.

II. Antecedentes

Una GPU puede ser vista como un conjunto de *compute units* (CU) que comparten un espacio de memoria global. Estas CUs son unidades de cómputo muy vectorizadas en las que el flujo de control se realiza mediante técnicas de predicación [13]. Cada CU contiene diversas unidades funcionales: unidades vectoriales, escalares, de salto, una interfaz a memoria global y una *local data share* (LDS) que provee de memoria local a los work-groups planificados en dicha CU. Los registros dentro de esta CU se encuentran alojados en las unidades vectoriales y escalares. Los registros vectoriales son utilizados, normalmente, para el cómputo de propósito general y son privados a cada work-item. Los registros escalares son compartidos por todos los work-items de un mismo wavefront y almacenan información común para ellos: índices de bucles, identificadores de work-group, máscaras de ejecución para la predicación, etcétera. Normalmente es el compilador el que decide utilizar estos registros escalares en lugar de los vectoriales como una optimización en el uso de recursos. Tanto los registros, como los recursos en la LDS y las unidades vectoriales, son divididos estáticamente al principio del cómputo. De esta forma se minimiza el coste de un cambio de contexto. La unidad LDS, encargada de manejar el acceso a memoria local, está compuesta por multiples bancos de memoria que proporcionan un gran ancho de banda. Cuando los work-items de un wavefront son planificados para acceder a la LDS, se detectan conflictos de banco y, en caso de existirlos, se serializa el acceso. De esta forma, en un instante dado, un banco de memoria local sólo da servicio a un work-item en concreto.

En este artículo realizamos una propuesta de TM hardware, por lo que se ha utilizado un simulador para implementar y evaluar los cambios necesarios en el hardware de las GPU existentes. En este caso, hemos escogido Multi2sim 4.2 [12]. Multi2sim es un simulador de CPU y GPU que incluye modelos de procesadores superescalares, multi-núcleo, y diversas arquitecturas GPU. Proporciona un simulador funcional, que ejecuta las instrucciones codificadas en un binario de una arquitectura concreta, y un simulador detallado, que proporciona la simulación de un *pipeline* completo dando como resultado una simulación a nivel de ciclo.

De entre las familias de GPU proporcionadas por el entorno de simulación se ha escogido la Southern Islands de AMD. Esta GPU proporciona una LDS de 64kb dividido en 32 bancos, de los cuales un work-group solamente puede direccionar 32kb. Cada work-group se divide en wavefronts de 64 work-items. Cada wavefront es planificado a una de las 4 unidades vectoriales disponibles, lo que proporciona soporte para un total de 4 wavefronts de 64 work-items cada uno (esto es, 256 work-items por work-group). Los wavefronts son las unidades planificadas dentro de la CU, y su planificación se hace de forma rotativa siguiendo una estrategia *round-robin*. Dentro de un wavefront, el control de flujo se realiza mediante predicación utilizando 2 máscaras de bits, ambas de 64 bit: EXEC y VCC. EXEC indica qué work-item está activo (bit a 1) o inactivo (bit a 0) dentro de un wavefront. VCC se actualiza en las operaciones aritméticas y de comparación e indica si el resultado ha sido 0. Su funcionamiento es el de un flag Z vectorizado. Los compiladores utilizan estas dos máscaras para implementar condicionales, saltos y bucles. Estas máscaras, comunes para todo un wavefront, se mapean en registros escalares.

III. TM hardware en Memoria Local

En esta sección presentamos un TM hardware que proporciona soporte para transacciones que sincronizan work-items de un mismo work-group utilizando para ello la memoria local. Habitualmente, el soporte TM en hardware requiere cambios significativos al hardware (introducir nuevas unidades funcionales), gran cantidad recursos de memoria para almacenar valores especulativos y la implementación de una alternativa software que garantice el progreso. Esta propuesta trata de minimizar los recursos hardware y de memoria necesarios, de forma que sea una solución fácilmente adoptable por los fabricantes. Además, proponemos un mecanismo de serialización por hardware que evite que los programadores tengan que escribir una alternativa software a la solución TM hardware para garantizar el progreso.

Nuestra propuesta extiende el ISA de la arquitectura con 2 nuevas instrucciones para marcar el comienzo y final de la transacción: *TX_Begin* y *TX_Commit*. Estas instrucciones pueden ser utilizadas por los compiladores para proporcionar sentencias de más alto nivel en lenguajes como OpenCL. El caso de transacciones anidadas se ha dejado como trabajo futuro y a ser resuelto por el hardware y el compilador. Dentro de una transacción, todas las operaciones sobre memoria local gestionadas por la LDS son consideradas transaccionales (esto es, no existen unas instrucciones de lectura y escritura explícitas *TX_Read* y *TX_Write*). Cuando un wavefront ejecuta la instrucción TX_Begin, sus work-items activos (es decir, los que tienen su bit correspondiente en EXEC puesto a 1) comienzan la transacción. El TM hardware propuesto sigue un esquema *eager*, por lo que, en cada acceso a memoria, se ejecuta un algoritmo de detección de conflictos y de gestión de versiones, y el modelo SIMT es informado de los posibles conflictos para prevenir que el work-item en conflicto continúe su ejecución. La ejecución de la instrucción TX_Commit marca el final de la transacción para el wavefront. Si ningún work-item de wavefront ha presentado conflictos, la transacción finaliza y la ejecución continúa por la siguiente instrucción. En el caso de que algún work-item presente un conflicto, la transacción debe ser reiniciada para dichos work-items saltando de nuevo a la instrucción TX_Begin. Aprovechamos el hecho de que, en un instante dado, cada banco de LDS es accedido únicamente por un work-item para implementar un mecanismo de detección de conflictos y gestión de versiones por banco de memoria. La figura 1 muestra los cambios realizados en cada banco de memoria local de la LDS para implementar la funcionalidad propuesta. En los siguientes apartados se explicarán los cambios al modelo de ejecución SIMT necesarios para gestionar la transacción, así como los mecanismos de detección de conflictos y gestión de versiones.

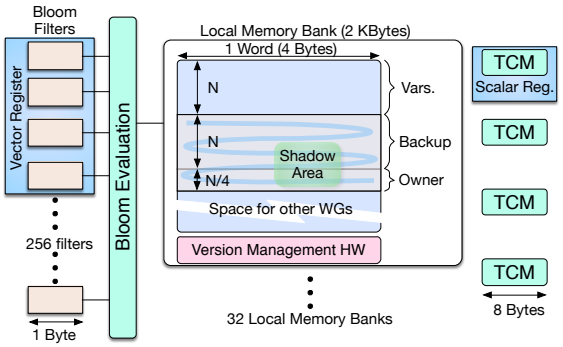

Fig. 1: Recursos hardware y de memoria propuestos para dar soporte TM hardware en los bancos de memoria local.

A. Modificaciones al Modelo de Ejecución SIMT

El modelo de ejecución SIMT se basa en la existencia de las máscaras EXEC y VCC. Cada wavefront mantiene una copia privada de estas máscaras, que tienen un bit por cada work-item en el wavefront. En el caso de la arquitectura estudiada, estas máscaras son de 64 bits y se alojan utilizando registros escalares. La máscara EXEC indica qué work-items dentro del wavefront están activos, mientras que VCC funciona como un flag Z, indicando el resultado de las instrucciones aritméticas y de comparación. Con estas máscaras, los compiladores implementan bucles, condiciones y saltos utilizando un mecanismo de predicación. Inicialmente, puede considerarse la reutilización de estas máscaras para implementar los mecanismos de control de flujo de una transacción como, por ejemplo, desactivar los work-items que han presentado un conflicto, o implementar los saltos correspondientes al reinicio de una transacción. Sin embargo, esto puede generar inconsistencias. Una instrucción condicional tipo *if-then* modifica la máscara EXEC para dejar activos únicamente los work-items que cumplen la condición, restaurando su valor tras la ejecución del bloque condicional. Si la máscara EXEC fuese modificada para desactivar un work-item que presenta un conflicto dentro del bloque condicional, éste seria re-activado (erróneamente) tras la ejecución de dicho bloque. Por este motivo, proponemos el uso de una nueva máscara *Transaction Conflict Mask* (TCM) por wavefront, mapeada en un registro escalar, que indique qué work-item ha presentado un conflicto en los accesos a memoria. La máscara TCM, de 64 bits, es inicializada a 0 por la instrucción TX_Begin. En un acceso a memoria, si se detecta un conflicto, se pone a 1 el bit de TCM correspondiente al work-item que ha presentado el conflicto. Si, al ejecutar la instrucción TX_Commit, TCM está a 0, significa que no ha habido ningún conflicto y, por tanto, la transacción se da por correcta y finalizada. Un 1 en alguno de los bits indica un conflicto. En este caso, la instrucción TX_Commit copia TCM en EXEC y provoca un salto a la instrucción TX_Begin. De esta forma, se reinicia la transacción sólo para aquellos work-items que presentaron conflictos. Sin

embargo, esto requiere que EXEC sea salvaguardado antes de ejecutar TX_Begin por primera vez, y restaurarlo tras salir de TX_Commit sin ningún conflicto. Para ello, hemos reservado por hardware un registro escalar, aunque también podría hacerse mediante software utilizando el compilador. Una importante ventaja de utilizar esta máscara TCM, completamente manipulada por hardware, es que los compiladores y el programador no necesitan hacer ningún cambio en la forma de implementar condicionales y bucles. Los únicos cambios potencialmente necesarios es promover el uso de registros vectoriales, privados a los work-items, en lugar de los escalares, compartidos por los work-items de un mismo wavefront. El motivo es que, si indice de un bucle es mapeado en uno de estos registros escalares y algunos work-items del wavefront progresan mientras que otros presentan conflictos, el valor que debe tomar este registro compartido puede quedar inconsistente. Por ello, los compiladores deben omitir la optimización que promociona el uso de registros escalares en lugar de vectoriales para bloques de código que se vayan a ejecutar dentro de una transacción.

La utilización de TCM no garantiza progreso: es posible que la instrucción TX_Commit deba reiniciar la transacción varias veces y se encuentre ante un bucle de reintentos infinitos debido a un conflicto en los accesos por parte de 2 o más work-items. Esto se detecta al comprobar que dos veces consecutivas se ha reiniciado la transacción sin ningún cambio en TCM, indicando que no ha habido progreso desde el último reintento. Estos conflictos pueden darse bien entre work-items del mismo wavefront, o bien entre work-items del mismo work-group, pero ubicados en diferentes wavefronts. Para garantizar el progreso, sin intervención del programador ni del compilador, proponemos una serialización de la ejecución en 2 niveles. En primera instancia, y dado que los work-items de un wavefront se ejecutan en *lockstep*, suponemos que el conflicto se ha dado dentro del wavefront. En el caso de 2 reintentos con el mismo valor en TCM, serializamos la ejecución de ese wavefront. En este caso, en lugar de reiniciar TCM a 0 en la instrucción TX_Begin, se sustituye únicamente uno de sus bits a 1 por 0. Esto es equivalente a reiniciar la transacción con un solo work-item. Llamamos a este mecanismo *wavefront serialization*. Si, aún así, en el siguiente reintento no existe progreso, se debe al conflicto con algún work-item de otro wavefront. En este caso, se provoca que todos los demás wavefronts aborten la transacción, y vuelvan a la instrucción TX_Begin, quedando detenidos en esta instrucción hasta que el wavefront actual ha alcanzado la instrucción TX_Commit. El wavefront actual debe adoptar el mismo mecanismo que en el caso de wavefront serialization. Cuando este wavefront termina el reintento de transacción, se permite el progreso del resto de wavefronts. Este segundo mecanismo se denomina *work-group serialization*. Para el funcionamiento correcto de estos dos mecanismos se debe garantizar que un work-item que no presenta conflictos con ningún otro es capaz de finalizar una transacción con éxito. Esto será garantizado por los mecanismos de gestión de versiones y detección de conflictos.

B. Gestión de Versiones

La gestión de versiones es el mecanismo que maneja los valores especulativos y definitivos de los accesos a memoria de una transacción. En primer lugar, debemos gestionar los valores que se encuentran en los registros vectoriales privados a cada work-item. Para ello, proponemos el uso de otros registros. Al iniciar una transacción, todos los work-items activos dentro del wavefront realizan una copia de los valores de sus registros en registros auxiliares que llamaremos *shadow registers*. La detección de un conflicto por parte de un work-item implica restaurar los valores de los shadow registers en los registros originales para reintentar la transacción con los valores adecuados.

La parte más importante y novedosa de la gestión de versiones reside en la gestión de los valores especulativos en memoria local. La gestión de versiones se realiza de manera distribuida en cada uno de los bancos de memoria local (32 bancos en el caso de la arquitectura estudiada). La distribución por bancos asegura que se pueda realizar una gestión de versiones en paralelo, puesto que los bancos son accedidos en paralelo por diferentes work-items. Además, esta solución es escalable y portable a otras arquitecturas con un número de bancos diferente. Cuando el programador define variables en el espacio de memoria local, el compilador las agrupa y aloja en una sección contigua de la memoria ubicada en la LDS. Para implementar el TM se requiere que el compilador aloje otra zona de memoria contigua a las variables de usuario, que denominaremos *shadow memory*. Esta shadow memory, de forma lógica, contiene pares <*owner, value*> que indican, para cada palabra de memoria, el work-item que ha accedido a dicha posición y el valor a restaurar en caso de detección de conflicto (esto es, se realiza una copia del valor antiguo al acceder a memoria, y se restaura dicha copia al detectar un conflicto, realizándose las escrituras a memoria de forma *eager* sobre la posición de memoria definitiva). Físicamente, y por eficiencia, los pares <*owner, value*> de la shadow memory se encuentra ubicados de forma diferente, como puede observarse en la figura 1. Suponiendo que las variables de usuario ocupan N palabras de memoria de 4 bytes, la shadow memory precisa N palabras del mismo tamaño adicionales para gestionar los valores antiguos, y N bytes adicionales para almacenar el identificador de work-item. Situando estos dos espacios de forma contigua a las variables de usuario de la forma indicada en la figura 1, podemos acceder a los valores antiguos y el identificador del work-item con el cálculo de 2 *offsets*. La copia del valor antiguo de una variable en la posición K se encuentra $K+N$ palabras de memoria más adelante y el work-item que la ha accedido ha almacenado su identificador K bytes después del área de resguardo. Esta forma de

gestionar las versiones de las variables asegura que todas las variables pueden ser accedidas dentro de una transacción. Esto garantiza que un work-item sin conflictos podrá progresar, puesto que podrá hacer copias de todos los accesos especulativos que lleve a cabo. Además el mecanismo basado en *offsets* es simple y requiere de pocas modificaciones en el hardware del banco de memoria en LDS. Como contrapartida, es un método que consume gran parte de los recursos de memoria, pudiendo afectar al tamaño máximo de las variables alojadas en memoria local.

C. Detección de Conflictos

El mecanismo de detección de conflictos registra cada acceso a memoria, determinando si tiene un conflicto con un acceso anterior a la mismo posición por parte de otro work-item. Para ello, proponemos un procedimiento de detección de conflicto distribuido, que opera en paralelo en cada uno de los bancos de memoria y que se desarrolla en 3 etapas:

1. **Detección de conflictos rápida:** Basada en filtros de Bloom [14]. Cada work-item mantiene, por cada banco de memoria, un filtro de Bloom para registrar sus accesos. Por simplicidad, en la detección de conflictos no se distinguen lecturas y escrituras. Inicialmente consideramos filtros de Bloom de 8 bits por cada work-item. Estos filtros pueden ser almacenados en registros vectoriales como se indica en la figura 1. Aprovechando la localidad espacial que poseen la mayoría de las aplicaciones que explotan paralelismo de datos en la GPU, hemos determinado el uso de una operación módulo para registrar los accesos. Cuando a un banco se le solicita la palabra de memoria K, se evaúan los filtros de Bloom de todos los work-items asignados a dicho banco. La evaluación consiste en comprobar el bit $K\,\%8$, operación que es simple y para la que proponemos un hardware que pueda hacerlo en paralelo con los 256 filtros de Bloom necesarios. Esta evaluación puede dar 3 resultados: 1) ningún positivo, 2) positivo únicamente en el filtro correspondiente al work-item actual o 3) positivo en un filtro diferente al work-item actual. El caso 1) significa que es un nuevo acceso. Se debe marcar el bit correspondiente en el filtro de Bloom del work-item actual. En el caso 2) no podemos determinar si es un acceso repetido a dicha posición de memoria o un nuevo acceso, puesto que no sabemos si el filtro ha devuelto un falso positivo. Este caso debe refinarse en la siguiente etapa. El caso 3) indica el posible acceso de otro work-item a dicha posición de memoria, lo que es considerado como un conflicto. En este último caso, la máscara TCM es actualizada colocando a 1 el bit correspondiente.
2. **Modificación de shadow memory:** Una vez conocido el resultado de la detección de conflicto, se añade un hardware para gestionar la shadow memory. Si la etapa anterior resultó en el caso 1) (nuevo acceso), se debe hacer una copia del valor en memoria y establecer la ID del work-item actual como dueño de dicha posición en la entrada correspondiente de la shadow memory. En el caso 2) se debe determinar si es un nuevo acceso, o si es un segundo acceso a la misma posición. Para ello, se examina la entrada correspondiente en la shadow memory. Si la entrada no está asociada al work-item actual, es un nuevo acceso y se procede como en el caso anterior. En otro caso, es un acceso repetido a la misma posición y no se requiere ninguna acción. Por último, el caso 3) indica un conflicto. En este caso, se deben restaurar las posiciones de memoria asociadas al work-item actual a sus valores originales, y se limpian sus filtros de Bloom.
3. **Comunicación del conflicto:** La primera etapa, detección de conflictos rápida, actualizó la máscara TCM y la segunda etapa, en caso de conflicto, restauró los valores modificados por el work-item en el banco de memoria actual. Sin embargo, los demás bancos de memoria podrían contener valores especulativos que no han sido restaurados. Debido a esto, esta última etapa lee la máscara TCM, que ha sido actualizada en paralelo por todos los bancos, y procede a restaurar los valores originales de las posiciones de memoria accedidas por work-items con conflicto y limpiar sus filtros de Bloom.

Debido a que se han introducido un mecanismo que distingue falsos positivos en el filtro de Bloom del work-item que accede a memoria, el progreso está garantizado cuando se ejecuta un work-item por wavefront, lo cual asegura que los mecanismos de serialización en wavefront y work-group funcionan adecuadamente.

IV. Evaluación

En este trabajo hemos presentado un TM hardware que permite definir transacciones sobre la memoria local compartida por los work-items pertenecientes a un mismo work-group. Normalmente, esta memoria es utilizada como *scratchpad* para almacenar estructuras de datos de forma temporal. Por ello, inicialmente hemos utilizado una tabla hash (HT) para evaluar nuestra propuesta, ya que es una estructura de datos conocida y utilizada por muchas aplicaciones. El trabajo futuro incluye la evaluación de otras estructuras de datos y algoritmos. En HT, cada uno de los 256 work-items pertenecientes al work-group trata de introducir su ID en una tabla hash. La tabla hash tiene un número fijo de entradas, pero cada entrada tiene capacidad suficiente para almacenar todos los posibles elementos. En la evaluación, el número de entradas se ha variado desde 2 (HT2), duplicándose, hasta 256 (HT256). El escenario HT2 es el que presenta mayor núnero de conflictos, ya que cada uno de los 256 work-items está tratando de insertar su ID teniendo únicamente 2 entradas disponibles para ello; mientras que el escenario HT256 no presenta ningún conflicto y todos los work-items son capaces de insertar su ID inmediatamente. La en-

trada se determina realizando la operación $ID\,\%N$, siendo N el número de entradas de la tabla. Una vez conocida su entrada, el work-item debe recorrer todos los elementos de la entrada asignada hasta encontrar un hueco disponible, e insertar en él su ID. Este proceso, en el que varios work-items potencialmente colisionan en la misma entrada, debe ser definido dentro de una transacción. La transacción comprende leer un elemento, comprobar si está vacío y, en ese caso, insertar el ID.

Se han implementado 3 versiones de este algoritmo: una versión utilizando TM por hardware, una versión en la que se ha serializado la execución de la transacción (TX. Serialization), fácilmente implementable pero menos eficiente, y, por último, una versión utilizando cerrojos de grano fino (FGL) que es más eficiente pero conlleva un esfuerzo de programación mayor. Dentro de una transacción, la latencia asignada a las operaciones de memoria local se ha medido en cada banco de memoria. Se ha estimado en 1 ciclo la evaluación de filtros de Bloom, y tantas veces la latencia de acceso a un banco de memoria como entradas de la shadow memory haya que modificar. Como la memoria local está compuesta de 32 bancos, la latencia total será igual a la del banco más lento. A las operaciones TX_Begin y TX_Commit se les ha asignado una latencia igual a las de una instrucción de tipo escalar. Las instrucciones escalares son aquellas que afectan a todo un wavefront, como puede ser una barrera o, en este caso, una transacción. La figura 2 muestra la evaluación realizada utilizando HT.

La figura 2.(a) muestra la aceleración conseguida por el uso de TM y FGL, tomando como base la serialización de la transacción. En ella se aprecia que, en casos favorables, nuestra solución TM alcanza el rendimiento proporcionado por FGL. En todos los casos, nuestra propuesta supera a la serialización de la transacción consiguiendo una aceleración entre 20X y 70X con un esfuerzo de programación similar.

En figura 2.(b) muestra un *breakdown* de la ejecución, en la que se ve que el coste en ciclos de utilizar una solución TM de estas características está en torno al 15 % del total de ciclos. Los escenarios con menor probabilidad de conflicto emplean un mayor porcentaje de ciclos liberando recursos de las transacciones terminadas, mientras que aquellos con mayor probabilidad de conflicto los emplean en la gestión de versiones durante los accesos a memoria.

Por último, la figura 2.(c) muestra el porcentaje de transacciones que deben ser serializadas de entre el total de transacciones. Debido a que no existe una sincronización fuerte entre los wavefronts de un mismo work-group, nunca se ha dado la serialización de work-group.

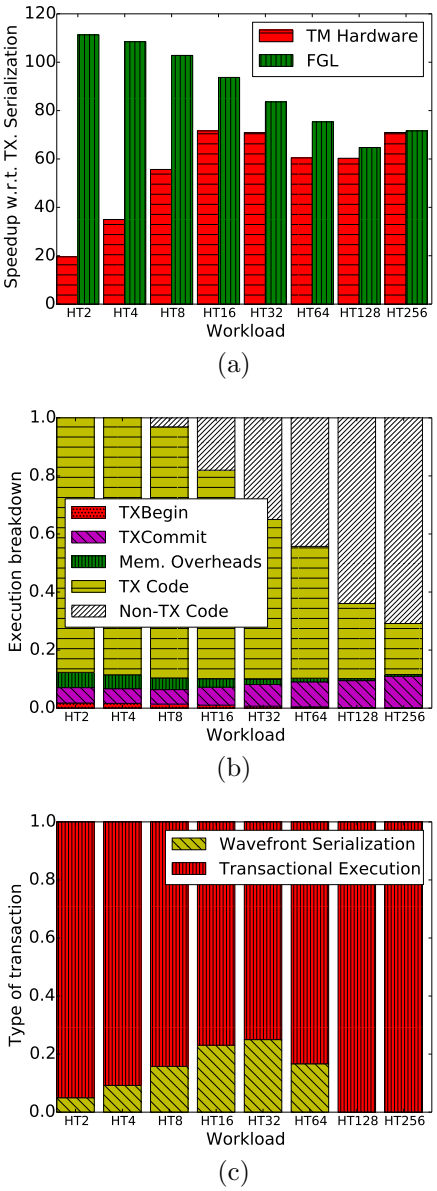

Fig. 2: Evaluación de la propuesta TM hardware utilizando HT.

V. Trabajo Relacionado

Existen otras propuestas recientes de TM sobre arquitecturas GPU.

En la vertiente software, encontramos los trabajos de Cederman *et al.* [5], Xu*et al.* [6] y Holey *et al.* [7]. Cederman *et al.* [5] proponen 2 soluciones que consideran transacciones entre *thread blocks* (work-groups) y no entre work-items. La solución de Xu *et al.* [6] proponen la primera solución a nivel de work-item que se fundamenta en una detección de conflictos en dos fases: una primera basada en *timestamp* (más rápida) y una segunda basada en valor (más precisa). Holey *et al.* [7] han propuesto 3 soluciones software TM. La primera realiza una detección de conflictos *eager*, basada en una estrategia de mantener un registro de los work-items que han leido cada variable y permitiendo un único work-item escritor. La segunda estrategia simplifica la primera y no distingue entre lecturas y escrituras. Señalar que, puesto que la segunda implementación les ha proporcionado mejores resultados que la primera a pesar de no distinguir entre lecturas y escrituras, nos hemos decantado por adoptar la misma estategia para implementar una solución simple y efectiva. La tercera

estrategia se basa en anotar, con un *timestamp*, el instante de modificación de cada variable; validando las lecturas al final de la transacción en lugar de en cada acceso. Ninguno de los 3 trabajos anteriores sobre TM software en GPU ha considerado el espacio de memoria local en su implementación.

Fung *et al.* [8], [9] han hecho la única propuesta TM hardware en GPU hasta la fecha. Su propuesta se basa en la creación de unas unidades especializadas para el *commit* de la transacción, que se hace de forma *lazy* al final (esto es, la detección de conflictos se lleva a cabo una vez alcanzada la instrucción TX_Commit, en lugar de en cada acceso de memoria). Para ello, cada work-item debe registrar sus accesos en unos *buffers* de escritura y lectura alojados en su memoria privada, pudiendo coexistir múltiples versiones privadas de la misma variable al mismo tiempo. Nuestra propuesta *eager* elimina la necesidad de almacenar estos buffers y de transmitirlos a las unidades de commit. Además, su trabajo contempla únicamente el espacio de memoria global. Nuestra propuesta, que es la única existente en considerar el espacio de memoria local, será extendida a memoria global en un trabajo futuro. De esta forma, podrá existir una comparación cuantitativa (y no sólo cualitativa) de ambas propuestas.

VI. Conclusiones y Trabajo Futuro

En este artículo hemos presentado una solución TM hardware para dar soporte a transacciones sobre la memoria local de arquitecturas GPU. Las principales carácterísticas de esta solución son una granularidad a nivel de work-item y la detección de conflicto y gestión de versiones distribuidas en los bancos de memoria. La solución propuesta es potencialmente capaz de igualar aquellas basadas en cerrojos de grano fino y supera a la serialización en el acceso a las secciones críticas. Además, se propone un mecanismo de serialización que evita que los programadores deban implementar una alternativa software para garantizar el progreso de la transacción.

El trabajo futuro evaluará de forma más exhaustiva la solución propuesta, además de incorporar optimizaciones en la gestión de memoria y planificación de work-items. En una etapa posterior, se incorporará al esquema propuesto el espacio de memoria global disponible en las arquitecturas GPU.

Agradecimientos

Este trabajo es parte de una colaboración entre el Departamento de Arquitectura de Computadores de la Universidad de Málaga y el grupo NUCAR de la Northeastern University en Boston, MA, USA. Los autores agradecen al profesor David Kaeli y al investigador Rafael Ubal de dicha universidad su colaboración.

Referencias

[1] NVIDIA, *NVIDIA CUDA Programming Guide.*

[2] Khronos, *The OpenCL Specification. Version 2.0.*

[3] Maurice Herlihy and J. Eliot B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93)*, 1993, pp. 289–300.

[4] Tim Harris, James Larus, and Ravi Rajwar, *Transactional Memory, 2nd*, Morgan & Claypool Publishers, USA, 2010.

[5] Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry, "Towards a software transactional memory for graphics processors," in *10th Eurographics Conf. on Parallel Graphics and Visualization (EG PGV'10)*, 2010, pp. 121–129.

[6] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian, "Software transactional memory for GPU architectures," in *Ann. IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'14)*, 2014, pp. 1:1–1:10.

[7] A. Holey and A. Zhai, "Lightweight software transactions on GPUs," in *43rd Int'l Conf. on Parallel Processing (ICPP'14)*, 2014, pp. 461–470.

[8] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt, "Hardware transactional memory for GPU architectures," in *44th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'11)*, 2011, pp. 296–307.

[9] Wilson W. L. Fung and Tor M. Aamodt, "Energy efficient GPU transactional memory via space-time optimizations," in *46th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'13)*, 2013, pp. 408–420.

[10] David A. Wood Daniel J. Sorin, Mark D. Hill, *A Primer on Memory Consistency and Cache Coherence*, Morgan & Claypool Publishers, 2011.

[11] Nuno Diegues, Paolo Romano, and Luís Rodrigues, "Virtues and limitations of commodity hardware transactional memory," in *23rd Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'14)*, 2014, pp. 3–14.

[12] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli, "Multi2Sim: A Simulation Framework for CPU-GPU Computing," in *21st Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'12)*, 2012.

[13] Carole Dulong, "The ia-64 architecture at work," *Computer*, vol. 31, no. 7, pp. 24–32, 1998.

[14] Burton H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, July 1970.