# Memoria Transaccional Software en Procesadores CPU+GPU Heterogéneos

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

*Resumen*— **En los procesadores multi-núcleo, la memoria transaccional (TM) ha aparecido como una alternativa prometedora a las técnicas basadas en cerrojos para garantizar exclusión mutua y está siendo incluida como parte de procesadores comerciales. De igual forma, dado que las GPUs se están convirtiendo en el acelerador más popular de la actualidad, los fabricantes están integrándolas dentro del mismo chip, creando las llamadas APUs (Accelerated Processing Units). Sin embargo, la sincronización entre CPU y GPU aún se lleva a cabo con mecanismos muy simples basados en operaciones atómicas y señales. Por tanto, es responsabilidad de los programadores implementar técnicas más avanzadas de exclusión mútua. Las técnicas basadas en TM aún no han sido explotadas en este tipo de procesadores y, por tanto, es importante hacer propuestas de sincronización avanzadas.**

**En este artículo proponemos una librería de TM software enfocada a su uso en procesadores APU. El objetivo es que las transacciones puedan ejecutarse tanto en CPU como en GPU simultáneamente y que se permita la sincronización en forma de exclusión mutua entre ambos dispositivos. Nuestra propuesta, llamada APUTM, se enfoca en minimizar la comunicación entre la CPU y la GPU de los metadatos requeridos para manejar TM. La evaluación de esta propuesta muestra que, utilizando este mecanismo de sincronización, es posible mejorar el tiempo de ejecución de las aplicaciones secuenciales con un reducido esfuerzo en la programación.**

*Palabras clave*— **Memoria transaccional, Procesadores heterogéneos.**

## I. Introducción

Hoy en día la programación multi-hilo es esencial para sacar el mayor provecho de los procesadores multi-núcleo, GPUs, y otros tipos de procesadores. En la programación multi-hilo, uno de los mayores problemas a los que se enfrentan los programadores cuando diseñan sus algoritmos es el acceso a datos compartidos: si dos o más hilos de ejecución tratan de acceder al mismo objeto compartido y, al menos uno de ellos va a realizar una modificación, los programadores deben implementar un mecanismo que garantice la exclusión mútua. Normalmente, estos mecanismos se basan en el uso de cerrojos. Sin embargo, el uso de cerrojos presenta varias desventajas como un bajo rendimiento (en el caso de cerrojos de grano grueso) o una alta dificultad en la programación (en el caso de cerrojos de grano fino).

La memoria transaccional (TM, por sus siglas en inglés) [13], [12] es una técnica prometedora, alternativa a los cerrojos, para garantizar exclusión mútua. La interfaz de TM permite al programador confinar la zona de exclusión mútua (sección crítica) dentro de una transacción. Las transacciones se ejecutan en paralelo y de forma especulativa y garantizan la atomicidad de sus operaciones. La exclusión mutua se garantiza gracias a unos mecanismos de detección de conflicto y de gestión de versiones. La detección de conflicto señala si 2 o más transacciones han accedido en paralelo al mismo objeto en memoria y, al menos uno de ellos, ha intentado modificarlo. En este caso, se dice que las transacciones tienen un conflicto y, algunas de ellas, abortan su ejecución y deshacen los cambios especulativos. Precisamente el mecanismo de gestión de versiones se encarga de mantener las estructuras de datos que permiten diferenciar entre cambios especulativos y los valores definitivos de los objetos en memoria. De esta forma, con TM se puede obtener un buen rendimiento a la vez que se proporciona al programador una interfaz simple y, por tanto, se facilita la programación de aplicaciones multi-hilo. TM está demostrando ser un mecanismo de sincronización muy valioso tanto en CPUs [12], [6], [8], [5], [7] como en GPUs [2], [3], [9], [10], [14], [18], [20]. Además, los fabricantes de CPUs están empezando a incluir TM en el hardware de sus procesadores: Intel TSX [21], IBM BlueGene/Q [19], System Z [15] y Power 8 [1].

Aparte de TM, la industria ha comenzado desde hace unos años a incluir GPUs integradas en sus CPUs. Comercialmente, este tipo de procesadores se suelen denominar APUs (Accelerated Processing Unit). Además, el consorcio Heterogeneous Systems Architecture (HSA)[1] está estableciendo una serie de estándares para facilitar la programación de este tipo de procesadores. En concreto, las APUs que siguen las especificaciones de HSA ofrecen un espacio de memoria unificado entre CPU y GPU, y operaciones atómicas (llamadas *platform atomics*) que permiten comunicar CPU y GPU. Con estas herramientas, los programadores están comenzando a distribuir sus aplicaciones entre CPU y GPU, y la necesidad de mecanismos de sincronización más avanzados comienza a surgir. Por tanto, es importante comprender la forma en la que los mecanismos de TM pueden adaptarse a los procesadores heterogéneos tipo APU.

Implementar una solución de TM en un procesador tipo APU no es una tarea trivial. La CPU y GPU trabajan con diferentes modelos de programación. Un procesador multi-núcleo sigue el modelo MIMD, mientras que la GPU sigue el modelo SIMD. Además ambos tienen especios de memoria diferentes. Aunque ambos dispositivos tengan acceso a la misma memoria física, acceden a través de un sistema de cache que puede estar implementado de forma diferente.

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

[1]`http://www.hsafoundation.com`

Además, la GPU tiene acceso a un espacio de memoria denominado memoria local (en nomeclatura OpenCL) o memoria shared (en nomenclatura CUDA) utilizado para acelerar la ejecución de las aplicaciones. Por último, la comunicación entre ambas plataformas se puede realizar utilizando las *platform atomics* mencionadas anteriormente. Habitualmente, estas operaciones son costosas en términos de latencia.

Para comprender y superar estas dificultades proponemos APUTM, una solución TM por software diseñada para sincronizar transacciones ejecutadas tanto en CPU como en GPU en un dispositivo heterogéneo tipo APU. APUTM puede ser configurado para usarse sólamente en CPU, sólamente en GPU, o en ambos dispositivos simultáneamente. APUTM presenta dos implementaciones diferentes, una en CPU y otra en GPU, pero con una interfaz común. La implementación de APUTM se basa en las ideas de NOrec [5], combinando un método de detección de conflictos rápido con otro más preciso en caso de ser necesario. Además, en GPU, APUTM se adapta al modelo SIMD, permitiendo que múltiples transacciones sin conflicto actualicen la memoria en paralelo. En la evaluación de APUTM analizamos distintas configuraciones y proporcionamos sugerencias para futuras mejoras.

## II. Antecedentes

### A. Memoria Transaccional

La memoria transaccional (TM) [13] se ha propuesto como un mecanismo eficiente para garantizar la exclusión mútua en programas multi-hilo. TM ofrece una interfas simple para manejar el acceso a datos compartidos. TM puede ser implementado en software, hardware o de forma híbrida. En este artículo nos enfoncamos en las soluciones software. Habitualmente, la interfaz de TM consta de 4 instrucciones: *TMBegin* y *TMCommit*, utilizadas para delimitar la sección crítica, y *TMRead* y *TMWrite* para señalar los accesos transaccionales a memoria.

Los dos puntos clave en la implementación de una solución TM son la gestión de versiones y la detección de conflictos. La gestión de versiones se encarga de mantener los datos especulativos que la transacción trata de escribir. Estos datos deben convertirse en definitivos si la transacción ejecuta la instrucción TMCommit sin haber encontrado ningún conflicto. En otro caso, la información especulativa tiene que ser descartada. La gestión de versiones puede ser implementada de forma *eager* o *lazy*. Una implementación eager permite a las transacciones escribir directamente en las posiciones definitivas de memoria, aunque previamente debe hacerse una copia del valor previo en un *undo-log* para que pueda ser restaurado en caso de conflicto. Las implementaciones lazy mantienen los valores especulativos en un *write-log* y no modifican la memoria hasta que la transacción termine. Si la transacción termina con éxito, entonces los valores almacenados en el write-log se copian de forma definitiva en memoria. La gestión de conflictos se encarga de señalar un conflicto si dos o más transacciones acceden al mismo objeto en memoria y, al menos una de ellas, realiza una modificación. En este caso, la transaccion señala un conflicto y una de ellas debe abortar, descartando los valores especulativos. El mecanismo de detección de conflictos también puede ser implementado de forma eager o lazy. En una implementación eager, en cada acceso a memoria se trata de determinar si existe un conflicto, abortando la transacción en el caso de que así sea. En las implementaciones lazy los accesos a memoria son registrados, pero no se detectan conflictos en ese momento. Cuando finaliza la transaccion, la operacion TMCommit está a cargo de analizar todos los accesos a memoria hechos por la transacción durante su ejecución y de señalar los conflictos que pueda tener con otras transacciones. En este artículo contribuimos con una solución lazy-lazy.

### B. Procesadores APU

Para explicar la arquitectura de los procesadores APU utilizamos como ejemplo el procesador AMD Kaveri A10-7850K, compatible con las especificaciones HSA, y que es el utilizado para la evaluación de nuestra propuesta. Este procesador integra una CPU de 4 núcleos y una GPU con 8 unidades de cómputo (CU). Los hilos de CPU se mapean en los núcleos de CPU. En la GPU, sus hilos (también conocidos como *work-items* en nomenclatura OpenCL) se agrupan en *work-groups* que se asignan a las distintas CUs. A su vez, los work-items dentro de un work-group se agrupan en *wavefronts*. En esta arquitectura, un wavefront consta de 64 work-items y un work-group se compone, como máximo, de 4 wavefronts. Los work-items de un mismo wavefront comparten el mismo contador de programa y se ejecutan en *lock-step*. Además, en cada CU se encuentra una memoria de baja latencia denominada memoria local, para ser utilizada a modo de *scratch-pad* y acelerar los accesos a memoria de la aplicación. En esta arquitectura, cada work-group tiene acceso a 32Kb de memoria local. De entre las características de HSA disponibles en este procesador cabe destacar el espacio de memoria virtual compartido entre CPU y GPU y la coherencia de memoria entre ambos dispositivos. De esta forma, los punteros pueden ser vistos por ambos dispositivos sin necesidad de copias explícitas. Además, también se proporcional operaciones atómicas entre ambos dispositivos y operaciones de semántica en los accesos a memoria (esto es, *memory ordering*).

### C. Trabajo Relacionado

La implementación de CPU de APUTM está inspirada en NOrec STM [5]. NOrec es una implementación de TM software que realiza la detección de conflicto en el momento de commit (lazy), comparando los valores leidos durante la transacción con los valores actuales de memoria. Si alguno de los valores leidos durante la transacción ha cambiado, entonces se señala un conflicto. Para acelerar este proceso, NOrec mantiene un cerrojo con un número

de secuencia codificado utilizado para serializar las transacciones que escriben. Al comienzo de la transacción este número de secuencia es registrado. Si, al final de la transacción, el número de secuencia no ha cambiado, entonces la transacción puede actualizar sus valores especulativos en memoria. En este caso, el número de secuencia se incrementa. Sin embargo, en caso de que el número de secuencia haya cambiado durante la ejecución de una transacción, entonces la transacción debe detectar conflictos comprobando sus accesos a memoria.

En cuanto a las soluciones TM para GPUs, GPU-STM [20] tiene muchas cualidades deseables para las APUs. Al igual que NOrec, GPU-STM utiliza el concepto de un cerrojo con número de secuencia, lo que facilita la integración de ambas soluciones. Además, GPU-STM permite que las transacciones sin conflicto dentro de un wavefront escriban en memoria en paralelo. Sin embargo, en GPU-STM esto se consigue mediante el uso de varios cerrojos manejados con operaciones atómicas. En nuestro diseño tratamos de minimizar el uso de operaciones atómicas y utilizamos otras técnicas para permitir la escritura en paralelo.

## III. APUTM

En esta sección presentamos APUTM, un TM por software diseñado específicamente para procesadores APU. APUTM permite sincronizar transacciones en CPU y GPU minimizando el uso de operaciones atómicas necesarias para ello. Esto se consigue utilizando un cerrojo global con un número de secuencia (inspirado por NOrec) compartido entre transacciones de CPU y GPU. APUTM implementa un mecanismo de gestión de versiones lazy, al igual que un mecanismo de detección de conflictos lazy. En las siguientes subsecciones discutimos la implementación de APUTM.

```
1 //Global metadata
2 atomic_int * gclock;
```

```
1 //Private metadata
2 struct pr_descr{
3   int snapshot;
4   int status; //RUNNING or ABORTED
5   <address,value> * reads;
6   <address,value> * writes;
7 };
```

```
1 //Wavefront metadata
2 struct wf_descr{
3   atomic_long * commit_mask;
4   atomic_int * leader_id;
5   <address,owner> * wf_writes;
6   atomic_int * next_write;
7 };
```

Fig. 1: Metadatos globales, privados y de wavefront requeridos para implementar APUTM.

### A. Metadatos

En la figura 1 hemos clasificado los metadatos requeridos para implementar APUTM. Estos metadatos están divididos en 3 categorias: globales, privados, y de wavefront.

Los metadatos globales son accedidos por todas las transacciones, tanto de CPU como de GPU. Por tanto, estos metadatos deben implementarse utilizando operaciones atómicas entre plataformas. La única variable de este tipo es el cerrojo con secuencia que llamamos, a partir de ahora, *gclock*.

Los metadatos privados son individuales a cada transacción y constan de 4 elementos. *snapshot* es el último valor de gclock en el que se comprobó que la transacción es consistente con memoria. *status* muestra el estado de la transacción (ejecutando, o abortada). *reads* contiene las direcciones de los accesos de lectura a memoria, y los valores que se leyeron. *writes* se utiliza para almacenar las escrituras especulativas a memoria.

Por último, los metadatos de wavefront son exclusivos de la parte GPU de APUTM. Cada wavefront tiene, de forma privada, sus propios metadatos de wavefront, que son compartidos por todos los work-items del mismo. *commit mask* indica, para los work-items de un mismo wavefront, aquellos que han abortado. Algunas operaciones que se detallarán a continuación deben realizarse por un único work-item, que viene designado por la variable *leader_id*. Todas los accesos de escritura del wavefront deben ser registrados para detección de conflictos. Esto se lleva a cabo en la variable *wf_writes* que es gestionada por *next_write*.

### B. Gestión de Versiones

La figura 2 muestra el pseudo-código que implementa APUTM, y que utilizamos para explicar la gestión de versiones. Al comienzo de la transacción, en la operacion TMBegin, tanto las transacciones CPU como las GPU inicializan sus conjuntos de lectura y escritura (líneas 2-3), establecen la transacción como ejecutandose (línea 4) y obtienen una copia de gclock (línea 5). En el caso de las transacciones GPU, se inicializan los datos compartidos por el wavefront (líneas 6-9). Las operaciones de escritura especulativa, TMWrite, registran sus accesos de escritura en writes (línea 56), o actualizan cualquier valor previo (línea 54). En el caso de las transacciones GPU también deben registrarse en los metadatos de wavefront (líneas 57-60). Las lecturas de memoria, TMRead, en primer lugar comprueban si es un acceso RAW (*read-after-write*) y, en ese caso, retornan el valor almacenado en writes (líneas 65-66). En otro caso, se debe comprobar la consistencia con memoria (esto se hace por corrección y se explicará más adelante) y, en caso de ser consistente, se accede a memoria y se registra la lectura en reads (líneas 68-73). En este caso, no hay diferencias entre las transacciones de CPU y las de GPU. Por último, durante la operación TMCommit, los valores especulativos se hacen definitivos si no se ha detectado ningún conflicto. Esto se hace escribiendo en memoria todos los valores de writes (líneas 19-20) de las transacciones que no tengan conflicto. Destacar que, en el caso de GPU, este

```
1 TMBegin(){
2  clear(reads)
3  clear(writes)
4  status=RUNNING
5  snapshot=gclock.atomic_load()
6  if (GPU){
7    next_write.atomic_store(0)
8    clear(commit_mask)
9  }
10 }

11 TMCommit()
12 {
13  if(GPU)
14    checkWFConflicts()
15  acquire(global_lock)
16  consistent= checkConsistency()
17  if(consistent){
18    if(!empty(writes)){
19      for(<addr, val> in writes)
20         mem[addr]=val
21      atomic_add(gclock, 1)
22    }
23  }else{
24    abort
25  }
26  release(global_lock)
27 }

28 checkConsistency(){
29  if(snapshot==gclock.atm_load())
30    return TRUE
31  do{
32    time=gclock.atm_load()
33    for(<addr, val> in reads){
34      if(mem[addr]!=val)
35        return FALSE
36    }
37  }while(time!=gclock.atm_load())
38  snapshot=time
39  return TRUE
40 }

41 acquire(lck){
42  if(GPU){
43   atomic_store(leader_id,th_id)
44   if(th_id==leader_id)
45    while(!atomic_cas(lck,0,1)){}
46  }
47  if(CPU){
48   while(!atomic_cas(lck,0,1)){}
49  }
50 }

51 TMWrite(addr, val)
52 {
53  if(contains(writes, addr){
54    update(writes, addr, val)
55  }else{
56    add(writes, addr, val)
57    if(GPU) {
58     p=atomic_add(next_write,1)
59     wf_writes[p]=<addr, tx_id>
60    }
61  }
62 }

63 TMRead(addr)
64 {
65  if(contains(writes, addr){
66   val=get(writes, addr)
67  }else{
68   consistent=
69      checkConsistency()
70   if(consistent){
71     val=mem[addr]
72     add(reads,addr,val)
73   }
74  }
75  return val
76 }

77 checkWFConflicts(){
78  for(<addr1,val> in writes{
79   for(<addr2,ownr> in wf_writes{
80     if(addr1==addr2 &
81        ownr!=th_id &
82        commit_mask(owner)==0){
83         commit_mask(th_id)=1
84         abort
85      }
86   }
87  }
88  for(<addr1,val> in reads{
89   for(<addr2,ownr> in wf_writes{
90     if(addr1==addr2 &
91        owner!=th_id &
92        commit_mask(ownr)==0){
93         commit_mask(th_id)=1
94         abort
95      }
96   }
97  }
98 }
```

Fig. 2: Funciones que implementan APUTM.

proceso se realiza en paralelo para las transacciones ejecutándose en un mismo wavefront.

### *C. Detección de conflictos*

La detección de conflictos se realiza de forma lazy, es decir, implementada como parte de la instrucción TMCommit. Dado que la CPU y la GPU operan con distintos modelos de programación, la implementación de la detección de conflictos es distinta para cada procesador.

**Transacciones CPU:** Las transacciones en CPU deben adquirir un cerrojo para poder comenzar la detección de conflictos (línea 15). Una vez adquirido, los cambios en memoria no están permitidos por parte de ninguna otra transacción y se procede a comprobar la consistencia de los datos accedidos por la transacción (línea 16). La comprobación de consistencia consiste en comparar si gclock ha sido modificado desde la última comprobación. Si no ha sido modificado, la transacción es consistente (líneas 29-30). En otro caso, la transacción debe comprobar si los valores registrados en reads siguen siendo consistentes (líneas 31-36). Si la transacción ha probado ser consistente, entonces se procede a volcar los valores de writes a memoria y a actualizar el valor de gclock (líneas 17-22). Si no es consistente, la transaccion aborta y debe ser reintentada. Por último, el cerrojo es liberado para permitir a otras transacciones poder realizar la detección de conflictos.

**Transacciones GPU:** En GPU las transacciones deben, en primer lugar, comprobar los conflictos dentro de un wavefront (líneas 13-14) y, a continuación, operar como las transacciones CPU. En primer lugar, comprobamos que dentro de un wavefront 2 transa-

ciones no tratan de modificar la misma posición de memoria (líneas 78-87). Esto se hace comprobando, en paralelo, cada uno de los conjutos writes privados de las transacciones con el conjunto wf_writes compartido. Si una transacción comprueba que otra intenta modificar la misma posición (líneas 80-81), y que dicha transacción no ha sido previamente abortada (línea 82), entonces señala un conflicto y se marca como abortada (líneas 83-84). Este mecanismo establece un orden de prioridad que garantiza el progreso: si varias transacciones han intentado modificar la misma posición de memoria, la primera en hacerlo puede continuar la ejecución, mientras que el resto aborta. Finalmente, se debe repetir este proceso para filtrar aquellas transacciones que han intentado modificar una posición que ha sido leida por otra transacción (líneas 88-97). El proceso es similar, pero utilizando la variable privada reads en lugar de writes. Una vez filtradas las transacciones que tienen conflicto dentro del mismo wavefront, el resto puede continuar con el mismo proceso seguido en la versión de CPU con una diferencia: la adquisición del cerrojo. Para evitar que todos los hilos del wavefront (hasta 64 en esta arquitectura) compitan simultáneamente por el mismo cerrojo, seleccionamos uno de los hilos como líder (línea 43). Esto se hace utilizando una operación atómica sobre el espacio de memoria local, la cual es mucho más eficiente que las operaciones atómicas entre plataformas. En este caso, el líder es el encargado de obtener y liberar el cerrojo (línea 45). Por simplicidad no se muestra el código para liberar un cerrojo, pero es similar a código utilizado para adquirirlo. Una vez la transacción líder ha adquirido el cerrojo, el proceso es similar al seguido por las transacciones CPU, aunque en este caso puede llevarse a cabo en paralelo por todos los hilos no abortados del wavefront.

### D. Corrección

Los cambios en memoria realizados por una transacción pueden dejar los conjuntos de lectura (reads) de otras transacciones inconsistentes. Estas transacciones siguen su ejecución con un estado de memoria inconsistente, lo cual afecta a la corrección del algoritmo. Por tanto, al igual que en NOrec [5], se debe comprobar que los valores en memoria siguen siendo consistentes en cada acceso de lectura (líneas 68-69). Esto garantiza la propiedad de corrección conocida como opacidad [11]. Puesto que esta operación es costosa, en nuestra evaluación estimamos el coste de mantener o no la opacidad, delegando en este segundo caso la corrección del algoritmo en el programador.

### E. Operaciones WAR

Las operaciones de escritura tras lectura (WAR) son comunes en muchos algoritmos. Por ejemplo, un algoritmo que trata de modificar una fila de una base de datos de forma atómica realizará una lectura de dicha fila para luego escribir una versión modificada de la misma. En el pasado, se han propuesto optimizaciones para este tipo de transacciones tanto en CPU [16] como en GPU [14]. En este trabajo incluimos una optimización basada en la propuesta de Holey *et al.* [14]. En primer lugar, y dado el patrón de acceso WAR, un conflicto puede detectarse antes si consideramos como conflicto la lectura de la misma posición de memoria por parte de 2 transacciones. En segundo lugar, las posiciones de memoria que se leen y escriben dentro de una transacción tendrán 2 entradas en los metadatos: una en el conjunto de reads y otra en el conjunto de writes. Esta duplicidad puede evitarse creando un único conjunto que almacene tanto lecturas y escrituras, ahorrando espacio de almacenamiento para las transacciones WAR. Durante nuestra evaluación, llamamos *unified-log* a una solución que combina ambos conjutos en uno solo y que detecta conflictos durante las operaciones de lectura.

## IV. Evaluación

### A. Metodología

Para evaluar nuestra propuesta utilizamos el procesador APU AMD Kaveri 7850K tal y como está descrito en la sección II. En la evaluación utilizamos el número máximo de hilos soportados por la arquitectura sin realizar *oversubscription*: 4 hilos en CPU y 2048 work-items en GPU. El sistema contiene 8 GB de memoria DDR3 a 1.6 GHz. Para acceder a las capacidades HSA soportadas por el procesador, se ha utilizado el lenguaje C++AMP con las opciones de compilación por defecto incluidas en ROCM 1.2[2]. Los resultados de todos los experimentos son el promedio de 10 ejecuciones.

### B. Caracterización de APUTM

Para caracterizar APUTM hemos creado un benchmark sintético que nos permite ajustar el número de lecturas y escrituras que realiza cada transacción: comparamos 2, 4, y 8 accesos a memoria por transacción. Estos accesos a memoria se realizan sobre un array de datos compartidos, siendo la probabilidad de conflicto del 5 %. Se han realizado experimentos de forma separada en CPU, GPU y de la APU al completo. En los casos de CPU y GPU por separado, se ejecutan 100000 transacciones, mientras que en el caso de la APU se ejecutan 50000 en CPU y otras 50000 en GPU de forma concurrente con una partición estática. Esta partición no es un requisito de APUTM, sino que se utiliza para comparar la eficiencia de ambas soluciones. La figura 3 muestra la evaluación de este benchmark sintético. En cuanto a las versiones ejecutadas, se ha evaluado el uso de gclock para acelerar la detección de conflictos junto con los conjuntos de lectura y escritura separados (Gclock-rw), o bien unificando ambos en uno único y permitiendo detectar conflicto en las operaciones de lectura (Gclock-u). Para evaluar la efectividad de gclock para acelerar la detección de conflictos se han creado 2 versiones equivalentes a las anteriores, pero sin utilizar gclock y directamente comprobando el

[2] `https://github.com/RadeonOpenCompute/ROCm`

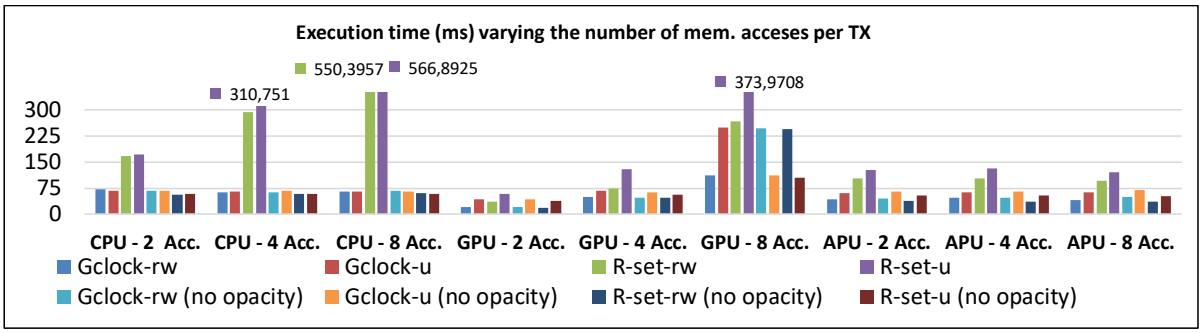



Fig. 3: Caracterización de APUTM para diferentes tamaños de los conjutos de lectura y escritura de las transacciones.

conjunto de lecturas (R-set-rw y R-set-u, respectivamente). Por último, para evaluar el coste de garantizar la corrección, se han evaluado otras 4 versiones, equivalentes a las anteriores, pero sin respetar la propiedad de opacidad.

De forma general, en la figura 3 observamos que utilizar gclock para acelerar la detección de conflictos tiene un impacto positivo tanto en CPU, GPU y la APU al completo, puesto que el tiempo de ejecución es menor. En la ejecución en CPU el impacto es mayor que en el resto de plataformas. Esto se debe a que en GPU (y, por tanto, en la mitad de las transacciones ejecutadas en la APU) varias transacciones ejecutan en *lockstep* debido al modelo de ejecución SIMD. Esto provoca que si una de las transacciones no logra aprovechar los beneficios de gclock, el rendimiento del resto de transacciones se ve afectado. Comprobamos también que, por lo general, el hecho de unificar los conjutos de lectura y escritura en un único *unified-log* no nos proporciona beneficios. En cuanto a las versiones sin opacidad, se consiguen mejores tiempos de cómputo, pero sin garantizar la corrección del algoritmo. Cabe señalar que, utilizando gclock, el impacto de garantizar la corrección es mínimo. Por tanto, la implementación más adecudada es la que utiliza gclock y opacidad.

### C. Estructuras de datos concurrentes

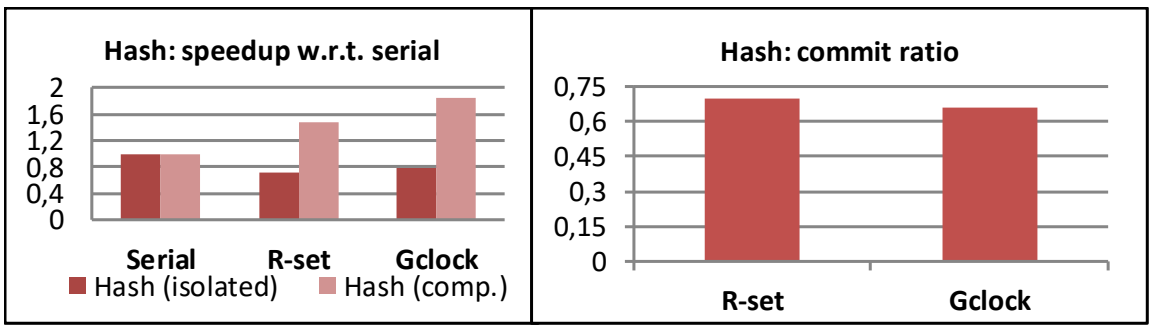
Para representar a las estructuras de datos concurrentes hemos implementado una tabla Hash utilizando APUTM. En este benchmark, hemos diseñado una tabla Hash con 50000 entradas en las que las transacciones intentan insertar valores. Cada transacción, una vez seleccionada la entrada de la tabla deseada, inserta un elemento al final de dicha entrada. La entrada está gestionada por un índice que debe incrementarse de forma atómica una vez se ha insertado el elemento. Estas transacciones son ejecutadas tanto por la CPU como por la GPU de forma concurrente.

La figura 4 muestra la evaluación llevada a cabo con la tabla Hash. En este caso, comparamos una versión de APUTM que utiliza gclock frente a la ejecución secuencial y otra versión que comprueba conflictos utilizando únicamente el conjunto de lecturas (R-set). Además se han evaluado 2 versiones de la tabla Hash: una únicamente con la inserción de elementos (isolated) y otra que simula una aplicación que realiza cálculos con los datos a insertar (computation). En la figura 4.a podemos observar como APUTM es capaz de mejorar el rendimiento de la aplicación cuando se introduce una componente de computación. Esto se debe a que el *overhead* de APUTM es aún importante y que necesita que la aplicación contenga una gran cantidad de cómputo para poder amorizarlo. Este *overhead* también influye en que el rendimiento no sea el esperado, y puede reducirse introduciendo optimizaciones o proponiendo mejoras arquitecturales. En la figura 4 observamos el ratio de transacciones que finalizan. Un valor de 1 indica que todas las transacciones finalizan en el primer intento, mientras que un valor de 0 indica que las transacciones nunca terminan (*deadlock*). En este caso comprobamos que ambas soluciones nos devuelven valores parecidos.

### D. Evaluación de Aplicaciones

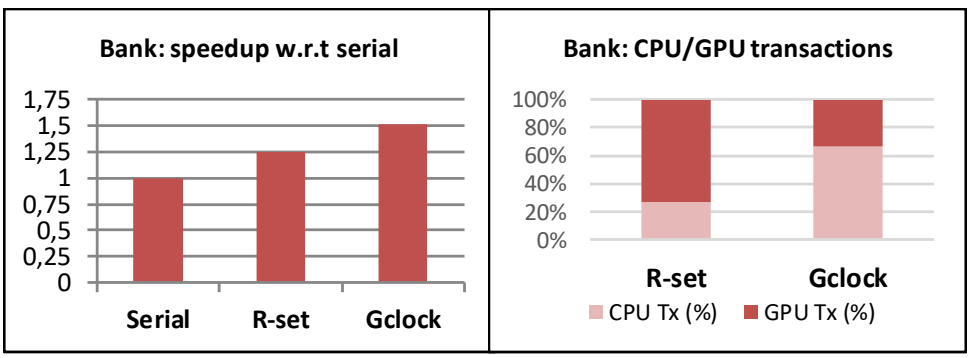
Para evaluar APUTM se han diseñado 2 apliaciones más: Bank, basada en la que se proporciona en TinySTM [8], y Genetic, que implementa un algoritmo genético para resolver el problema de la mochila.

La aplicación Bank simula transferencias entre distintas cuentas bancarias. Cada transferencia debe restar una cantidad de una cuenta bancaria e ingresarlo en otra. Las cuentas bancarias son escogidas al azar de entre un conjunto de 1 millón de cuentas, En este problema tenemos un conjunto de 10000 transferencias bancarias (10000 transacciones). Cada hilo de ejecución, siempre que queden transacciones pendientes, escoje una transacción y la ejecuta. De esta forma se realiza un reparto dinámico de la carga de trabajo. La figura 5 muestra la evaluación de este benchmark. En la figura 5.a observamos la aceleración de gclock y R-set (estos términos tienen el mismo significado que en el benchmark anterior) respecto a la versión secuencial del algoritmo. Observamos que el uso de gclock proporciona un mayor rendimiento, por lo que su uso es adecuado para acelerar las aplicaciones. En la figura 5.b observamos como se ha distribuido la ejecución de transacciones,



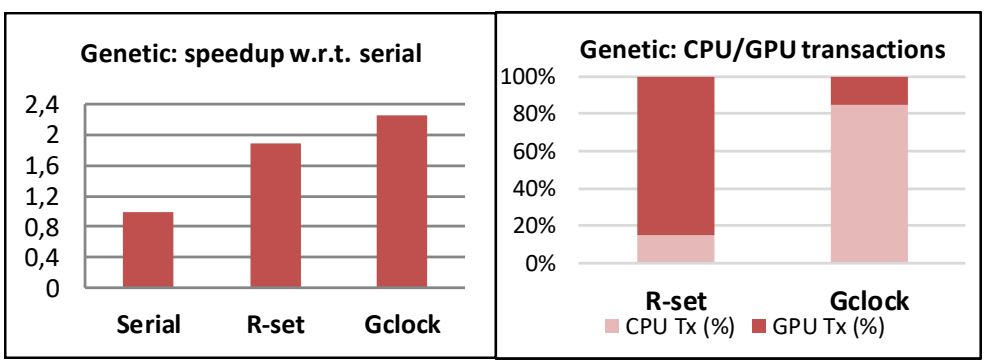

a b

Fig. 4: Evaluación de una tabla Hash con APUTM.




a b

Fig. 5: Evaluación de la aplicación Bank.




a b

Fig. 6: Evaluación de la aplicación Genetic.

puesto que en este caso las transacciones se distribuyen dinámicamente entre CPU y GPU. Podemos observar que, utilizando gclock, se ejecutan más transacciones en CPU que en GPU. Esto se debe a que, como hemos comprobado en el benchmark sintético, la escalabilidad de gclock es mayor en CPU y, por tanto, más transacciones son planificadas en ella. No ocurre lo mismo si omitimos el uso de gclock. En el escenario R-set se planifican más transacciones en GPU debido a su mayor potencia de cómputo y a que, en esta ocasión, la CPU no se beneficia del uso de gclock.

La aplicación Genetic implementa un algoritmo genético para resolver el problema de la mochila. Dado un conjuto de objetos de un peso determinado, y una mochila con una capacidad máxima definida, este algoritmo trata de averiguar qué objetos pueden colocarse en la mochila para llenarla lo máximo posible. Inicialmente, el algoritmo genera un conjunto de soluciones aleatorias (10 millones en este caso). Cada transacción selecciona 2 soluciones al azar. La solución más cercana al objetivo se devuelta al conjunto, mientras que la otra es modificada mediante un operador de mutación antes de ser devuelta. El uso de transacciones es necesario para evitar que 2 o más hilos de ejecución modifiquen la misma solución en

paralelo. En este benchmark, evaluamos 10000 iteraciones del algoritmo. Cada iteración se ejecuta dentro de una transacción que, al igual que en Bank, se asigna dinámicamente a la CPU o la GPU. La figura 6 muestra la evaluación de esta aplicación. En la figura6.a observamos la aceleración de R-set y gclock respecto a la versión secuencia. Al igual que en la aplicación anterior, se observa que gclock proporciona mayor escalabilidad. Respecto al número de transacciones ejecutadas en cada dispositivo (figura 6.b), comprobamos que se repite el mismo patrón. Si se utiliza gclock para acelerar la detección de conflicto, entonces se planifican más transacciones en CPU puesto que se beneficia de este mecanismo en mayor medida. Omitiendo el uso de gclock, la mayor parte de las transacciones son planificadas en GPU ya que ofrece mayor potencia de cómputo.

## V. Conclusiones y trabajo futuro

En este artículo presentamos APUTM, un sistema de memoria transaccional por software que, mediante el uso de transacciones, implementa exclusión mutua para procesadores tipo APU. Se han presentado varias implementaciones de las que deducimos que el uso de gclock consigue mejores tiempos de cómputo, lo cual permite implementar opacidad sin que afecte al rendimiento en gran medida.

Como líneas de trabajo futuro podemos señalar algunas optimizaciones que pueden aplicarse a APUTM para mejorar su rendimiento. En GPU, el acceso a las variables privadas puede optimizarse para que sigan un patrón *coalesced*, permitiendo a varios hilos acceder en paralelo a memoria. Además, podría estudiarse utilizar una aproximación de grano fino de gclock. Por ejemplo, cada objeto en memoria puede tener su propio cerrojo para permitir la detección de conflictos en paralelo, siempre y cuando sea sobre distintos elementos. Por último, sería interesante estudiar algún mecanismo de planificación para distribuir de forma más inteligente el número de transacciones ejecutadas en cada dispositivo.

## Referencias


[1] A. Adir, D. Goodman, et al. Verification of transactional memory in Power 8. In *51st Ann. Design Automation Conf. (DAC'14)*, pages 1–6, 2014.

[2] D. Cederman, P. Tsigas, and M. T. Chaudhry. Towards a software transactional memory for graphics processors. In *10th Eurographics Conf. on Parallel Graphics and Visualization (EG PGV'10)*, pages 121–129, 2010.

[3] S. Chen and L. Peng. Efficient GPU hardware transactional memory through early conflict resolution. In *22nd Int'l. Symp. on High Performance Computer Architecture (HPCA'16)*, 2016.

[4] L. Dalessandro and M. L. Scott. Strong isolation is a weak idea. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'12)*, 2012.

[5] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[6] D. Dice, O. Shalev, and N. Shavit. *Transactional Locking II*, pages 194–208. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[7] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[8] P. Felber, C. Fetzer, T. Riegel, and P. Marlier. Time-based software transactional memory. *IEEE Transactions on Parallel & Distributed Systems*, 21:1793–1807, 2010.

[9] W. W. L. Fung and T. M. Aamodt. Energy efficient GPU transactional memory via space-time optimizations. In *46th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'13)*, pages 408–420, 2013.

[10] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. Hardware transactional memory for GPU architectures. In *44th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'11)*, pages 296–307, 2011.

[11] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[12] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Ed.* Morgan & Claypool Publishers, USA, 2010.

[13] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural support for lock-free data structures. In *20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93)*, pages 289–300, 1993.

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

[15] C. Jacobi, T. Siegel, and D. Greiner. Transactional memory architecture and implementation for IBM System z. In *45th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'12)*, pages 25–36, 2012.

[16] W. Ruan, Y. Liu, and M. Spear. Transactional read-modify-write without aborts. *ACM Trans. Archit. Code Optim.*, 11(4):63:1–63:24, Jan. 2015.

[17] Q. Shen, C. Sharp, W. Blewitt, G. Ushaw, and G. Morgan. *PR-STM: Priority Rule Based Software Transactions for the GPU*, pages 361–372. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[18] A. Villegas, R. Asenjo, A. Navarro, O. Plata, R. Ubal, and D. Kaeli. *Hardware Support for Scratchpad Memory Transactions on GPU Architectures*, pages 273–286. Springer International Publishing, Cham, 2017.

[19] A. Wang, M. Gaudet, P. Wu, J. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of BlueGene/Q hardware support for transactional memories. In *21st Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'12)*, pages 127–136, 2012.

[20] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian. Software transactional memory for GPU architectures. In *Ann. IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'14)*, pages 1:1–1:10, 2014.

[21] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'13)*, pages 19:1–19:11, 2013.