Solapamiento transparente de tareas de comunicación y computación para mejor rendimiento de aplicaciones de GPU

Conference Paper · September 2018

DOI: 10.5281/zenodo.11213461

CITATIONS
0

READS
2

4 authors, including:

Yuri Torres
Universidad de Valladolid
39 PUBLICATIONS **256** CITATIONS
SEE PROFILE

Arturo Gonzalez-Escribano
Universidad de Valladolid
147 PUBLICATIONS **807** CITATIONS
SEE PROFILE

Diego R. Llanos
Universidad de Valladolid
158 PUBLICATIONS **922** CITATIONS
SEE PROFILE

Solapamiento transparente de tareas de comunicación y computación para mejor rendimiento de aplicaciones de GPU

Ismael J. Taboada[1] , Yuri Torres[2], Arturo Gonzalez-Escribano[2] y Diego R. Llanos[2]

Resumen— **El uso de aceleradores hardware de alto rendimiento, tales como las unidades de procesamiento gráfico (GPUs), ha ido en creciente aumento en los sistemas de supercomputación. Esta tendencia en fácilmente apreciable en la lista de computadoras mostradas por la clasificación TOP500. Programar este tipo de dispositivos es una tarea costosa que requiere un alto conocimiento sobre la arquitectura de cada uno de los aceleradores. Esta dificultad aumenta cuando se pretende explotar, de forma eficiente, los diferentes recursos hardware de un dispositivo. Este trabajo propone un modelo de programación que permite el solapamiento de tareas de comunicación y computación en dispositivos GPU mejorando así, el rendimiento de las aplicaciones. Nuestro estudio experimental muestra que este modelo oculta, de forma transparente, las latencias de comunicación si hay suficiente carga de comunicación, obteniendo hasta un 61.10 % de mejora de rendimiento comparado con nuestra implementación síncrona.**

Palabras clave— **Computación de alto rendimiento, Concurrencia, CUDA, GP-GPU, Modelos de Programación paralelos, Solapamiento de tareas.**

I. Introducción

El uso de aceleradores hardware de alto rendimiento tales como, las unidades de procesamiento gráfico (GPUs), ha ido en creciente aumento en los sistemas de supercomputación. Esta tendencia en fácilmente apreciable en la lista de computadoras mostradas por la clasificación TOP500 [1]. Estos co-procesadores están diseñados para explotar el paralelismo inherente de una aplicación y se utilizan de forma significativa para computación de propósito general (GPC – General Purpose Computing).

La diversidad de modelos y arquitecturas de dispositivos GPU hace que las soluciones diseñadas se deban implementar pensando en el hardware específico donde van a ser ejecutadas. Por ello, el desarrollador necesita unos conocimientos profundos sobre la arquitectura subyacente del dispositivo. Además, es necesario aplicar conocimientos sobre las librerías de desarrollo existentes para cada dispositivo, las cuales se encuentra en constante evolución. El uso de técnicas como el solapamiento de tareas de comunicación y computación [2], para mejorar el rendimiento de estas aceleradoras exigen tareas de desarrollo tediosas para el usuario y tendentes a fallos.

CUDA es una arquitectura y un modelo de programación de propósito general que ofrece una capa de abstracción sobre dispositivos GPU de NVIDIA [3,4]. OpenCL [5] es un modelo de programación paralelo que soporta dispositivos de diferentes fabricantes (Intel, NVIDIA, AMD). Ambas librerías exigen conocer las especificaciones de los dispositivos sobre los que se va a trabajar si se quiere hacer un uso eficiente, en la medida de lo posible, de los recursos hardware de los mismos.

Hitmap [6] y Controller [7], son librerías desarrolladas por el grupo de investigación Trasgo [8] que permiten una implementación independiente del acelerador, asegurando así, la adaptabilidad de la solución sobre cualquier máquina de forma transparente para el usuario. El usuario tiene la posibilidad de utilizar recursos específicos del dispositivo a través de la inclusión de código CUDA en las funciones paralelas.

Este trabajo propone un modelo de programación que permite el solapamiento de tareas de comunicación y computación en dispositivos GPU de NVIDIA. Utilizamos las librerías Hitmap y Controller para crear un prototipo que nos permita validar el modelo propuesto. Además, se presenta un estudio experimental que muestra cómo el modelo propuesto mejora el rendimiento general de las aplicaciones.

El resto del trabajo se organiza en las siguientes secciones: La sección II presenta algunos trabajos relacionados. La sección III introduce los conceptos básico de las librerías Hitmap y Controllers. La sección IV muestra las especificaciones del modelo propuesto. La sección V describe los detalles relacionados con la implementación. La sección VI expone los resultados experimentales obtenidos. Por último, la sección VII resume las conclusiones obtenidas y los problemas que abordaremos como trabajo futuro.

II. Trabajo relacionado

Existen tres aproximaciones para gestionar la coordinación de dispositivos heterogéneos. La primera aproximación es diseñar de forma manual soluciones utilizando modelos paralelos nativos (incluyendo aquellos provistos por el fabricante o diseñados específicamente para la explotación del conocimiento de la arquitectura) como son las librerías CUDA y OpenCL. Existen algunos frameworks como los presentados en [9, 10], que hacen un uso conjunto de herramientas nativas y los modelos bien conocidos OpenMP [11] y MPI [12].

Otra aproximación más abstracta se encuentra en el uso de herramientas, librerías, y estrategias orientadas en la ejecución paralela de bucles de forma automática en dispositivos heterogéneos, tales como

[1]Univ. Valladolid, e-mail: ismaeljose.taboada@alumnos.uva.es.
[2]Dpto. de Informática, Univ. Valladolid, e-mail: {yuri.torres|arturo|diego}@infor.uva.es.

LogFitc [13] y Maat [14]. Estas propuestas dividen las iteraciones de un bucle en tareas utilizando técnicas para movimiento asíncrono de datos, y ejecutando dichas tareas en los diferentes dispositivos. Sin embargo, estos trabajos no soportan la gestión de árboles de dependencias entre tareas generadas por bucles anidados.

Otros sistemas más abstractos y genéricos de programación, como GDM [15] y VAST [16], utilizan, de forma transparente, un conjunto de funcionalidades de GPU para cierto tipos de problemas y de estructuras de datos. Sin embargo, el uso de estas abstracciones no supone una mejora en la eficiencia del uso de los recursos del dispositivo en todos los casos.

III. Conocimiento previo

Esta sección introduce conceptos básicos de las librerías Hitmap [6] y Controller [7]. Además, se describe la arquitectura y las operaciones básicas de transferencia de datos de la implementación previa de Controller a través de un código ejemplo.

Hitmap es una librería usada en *Controller* que aporta una interfaz para el manejo de datos sobre kernels genéricos y portables. Un estructura *HitTile* es un manipulador que contiene meta-datos. Esta estructura guarda las direcciones de memoria donde se alojan los datos, tanto para la máquina anfitriona como para el dispositivo GPU.

La entidad Controller ofrece una metodología de programación sistemática con importantes características, como: (1) mecanismo para definir diferentes implementaciones de un kernel bajo un mismo nombre; desde kernels comunes que pueden ser reutilizados por distintos tipos de máquinas, hasta kernels específicos programados para un grupo concreto de dispositivos; (2) mecanismo transparente para administración de memoria, incluyendo comunicaciones de estructuras de datos entre el computador principal y sus correspondientes copias en las aceleradoras hardware; (3) sistema de mejora para seleccionar la configuración óptima en el lanzamiento de kernels (como la geometría de los bloques de hilos), guiada por una caracterización del kernel dada por el usuario con una sintaxis simple en la definición del kernel.

En Controller existen las siguientes peticiones de transferencia de datos: (1) *attach*, enlace de una estructura de datos a la instancia de Controller, con reserva de recursos y envío de la estructura de datos del anfitrión al dispositivo; (2) y *detach*, recuperación de una estructura de datos, y liberación de recursos de la instancia del Controller. La función detach de Controller es una operación que bloquea la ejecución de la aplicación hasta que los datos de la transferencia estén disponibles y los recursos se hayan liberado.

Un kernel se define mediante la primitiva `KERNEL_<type>`, donde `type` puede estar vacío para indicar que este pueda ser utilizado por diferentes tipos de dispositivos, o una identificador que indica el tipo de dispositivo asociado al código específico del kernel. Actualmente, la librería soporta los identificadores `GPU` para código CUDA y aceleradoras GPU de NVIDIA, `CPU` para código ejecutado sobre los cores de la máquina anfitriona, `XPHI` para co-procesadores Intel XeonPhi, `GPU_WRAPPER` y `CPU_WRAPPER` para código ejecutable en la máquina anfitriona, que incluye llamadas a librerías especializadas de GPU o CPU, como cuBLAS o MKL.

La figura 1 desarrolla un ejemplo de uso de las librerías Controller y Hitmap. Esta figura muestra el código de una implementación del método Jacobi-2D para un número determinado de iteraciones. En cada iteración las estructuras de datos son recuperadas por la máquina anfitriona desde el dispositivo GPU.

En la línea 2 de la figura 1 se muestra la definición de un kernel genérico. El kernel especifica el algoritmo a ejecutar mientras que Controller, automáticamente lo adaptará para que pueda ser ejecutado en cualquiera de los dispositivos soportados (GPU, XeonPhi o procesadores multi-core). La declaración del kernel contiene información de cada uno de los parámetros que va a tener mediante un conjunto de ternas. Cada terna incluye tipo, nombre y rol (parámetro de entrada o de salida).

Para implementar un kernel dentro de una aplicación es necesario seguir la siguiente estructura de desarrollo: (1) creación de la entidad de Controller asociada a un dispositivo concreto; (2) asociar cada estructura de datos HitTile a una instancia del Controller; (3) asociar la ejecución de un kernel con la misma instancia de Controllers que se utilizó para sus parámetros HitTile; (4) recuperar y liberación de las estructuras de datos HitTile con el mismo Controller.

El modelo de Controller implementa dos políticas diferentes de transferencia de datos: (1) *eager*, los movimientos de datos se realizan siempre que haya una petición de transferencia; (2) *lazy*, los movimientos de datos se realizan únicamente cuando un kernel haya modificado una estructura de datos. En ambos casos las operaciones se realizan de forma sincrónica garantizando la semántica del modelo.

IV. Propuesta

En esta sección describimos los mecanismos utilizados para detectar la dependencia de datos entre tareas de comunicación y computación, y explotar así las transferencias de datos asíncronas entre la máquina anfitriona y el dispositivo GPU.

A. Arquitectura de Controller

La figura 2 representa la nueva arquitectura propuesta para librería de Controller. Dicha arquitectura contiene una cola de peticiones, que incluyen la secuencia de lanzamientos de kernel y las peticiones de transferencia de datos. El modelo propuesto añade dos operaciones de transferencia de datos. Estas nuevas operaciones ayudan en la actualización de los datos entre su estado en el espacio de memoria de la máquina anfitriona y la de la aceleradora.

En adelante, denominaremos como *tareas* a las operaciones contenidas en la cola de peticiones.

```
/* Jacobi iteration: Generic kernel code for any type of device */
KERNEL(jacobi, 2, OUT, HitTile_float, A,  IN, HitTile_float, B )
{
    int x = thread.x + 1; int y = thread.y + 1;
    hit( A, x, y ) = ( hit( B, x-1, y ) + hit( B, x+1, y ) +
                       hit( B, x, y-1 ) + hit( B, x, y+1 ) ) / 4;
}
/* Host program using the Controller library */
int main()
{
  /* Stage 1: Controller creation, associated to the first GPU in the system */
  Cntrl comm;
  CntrlCreate(&comm, CNTRL_GPU, 0);

  /* Stage 2: Data structures creation and initialization */
  HitTile_float A = hitTile( float, 2, SIZE, SIZE );
  HitTile_float B = hitTile( float, 2, SIZE, SIZE );
  initMatrices(&A, &B);

  /* Stage 3: Data structures attachment: Associate to the Controller context */
  CntrlAttach(&comm, &A);
  CntrlAttach(&comm, &B);

  /* Stage 4: Kernel launchings with a number of logical fine-grain threads */
  Thread threadsSpace = ThreadInit( SIZE - 1, SIZE - 1);
  for( int iter=0; iter<NUM_ITERS; iter++ )
  {
    CntrlLaunch(comm, jacobi, threadsSpace, 2, &A, &B);
    CntrlMoveFrom(comm, &A);
    usePartialResultInHost(A);
    swap(A,B);
  }

  /* Stage 5: Data structures detachment, and free device resources */
  CntrlDetach(&comm, &A);
  CntrlDetach(&comm, &B);
  CntrlDestroy(&comm);
}
```

Fig. 1
Definición de kernel y programa principal de una implementación stencil del método Jacobi usando la librería Controller.

B. *Modelo de tareas asíncronas*

Este apartado describe la idea principal del solapamiento de tareas de comunicación y computación del modelo propuesto, mediante el caso que muestra la figura 2. En este ejemplo, el usuario implementa un programa paralelo usando las operaciones habituales de Controller (parte izquierda de figura 2). Este ejemplo contiene cuatro tareas principales, dos de ellas implican transferencias, y las otras dos, lanzamientos de kernel. La instancia de Controller almacena, en la cola, las tareas especificadas por el usuario, siguiendo el orden de inclusión. El primer kernel tiene A como entrada. Por ello, la ejecución de este kernel necesita ser sincronizada con el final del transferencia de A al dispositivo. La estructura de datos B es usada como entrada del segundo kernel, pero no es salida del primer kernel. Por consiguiente, la transferencia de B se puede solapar con la ejecución del primer kernel (parte central derecha de la figura 2). Sin embargo, la ejecución del segundo kernel tiene que sincronizarse con el final de la transferencia al dispositivo de B.

Este ejemplo nos sirve para introducir una primera aproximación sobre qué tareas pueden solaparse: (1) las tareas de lanzamiento de kernel son ejecutadas de forma sincrónica entre sí, manteniendo el orden en el que están almacenadas en la cola; (2) una tarea de lanzamiento de kernel debe ser sincronizada, esperando a cualquier comunicación que afecte a los parámetros del kernel; (3) una tarea de comunicación de datos de actualización puede solaparse con ejecuciones previas de kernels siempre que no sea usada como parámetro de este kernel.

Para implementar esta política, una instancia de Controller almacena información sobre las comunicaciones pendientes (aquellas que han sido iniciadas pero aún no se han completado). La instancia de Controller implementa una cola FIFO para el almacenamiento de las tareas que se incluyen desde el código del anfitrión. De esta cola, la instancia del Controller extrae las tareas y las procesa. En la evaluación de una tarea, la instancia de Controller se encontrará con los escenarios posibles:

1. Se evalúa una tarea de kernel cuya lista de parámetros tiene alguna intersección con la lista de comunicaciones pendientes, bloquea la extracción de tareas de la cola hasta que las comunicaciones necesarias hayan finalizado. Si la intersección fuese vacía, la tarea de kernel es movida a la cola de tareas listas.
2. Se evalúa una tarea de comunicación, el controlador evalúa si la estructura de datos aparece en la lista de parámetros del kernel que actualmente se está ejecutando, o en cualquier otro lanzamiento de kernel que esté dentro de la cola de preparados. Si la estructura de datos se encuentra en la lista de parámetros de algún kernel, el Controller bloquea la extracción de tareas y espera hasta que todos los kernels que involucren la estructura de datos finalicen. Cada vez que un kernel finaliza, cada uno de sus parámetros es marcado como disponible si ningún otro kernel en la cola de preparados lo utilizara posteriormente. Si el Controller estuviera esperando por una operación de comunicación que corresponda a una estructura de datos que esté disponible, esta operación se continuará, iniciándose de forma asíncrona. Si no se encuentra, la comunicación comienza de forma asíncrona y se anota como pendiente.

En el ejemplo de la figura 2, el Controller puede realizar las tareas (2) y (3) de forma concurrente

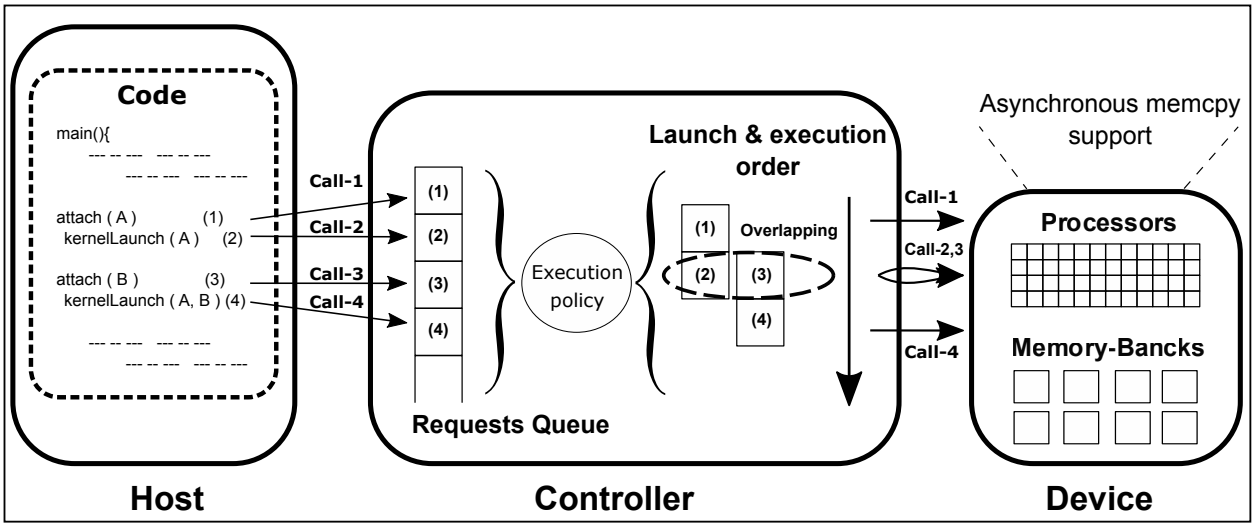

Fig. 2

Diagrama del modelo de arquitectura propuesta. Las peticiones de movimientos asíncronas son añadidas a una cola. La entidad del Controlador será la encargada de la ejecución de estas transferencias y las ejecuciones de los kernels.

en el dispositivo. La tarea (4) se despacha cuando las variables A y B han sido transferidas de forma completa a la memoria global del dispositivo (fin de comunicaciones previas), y el kernel previo haya finalizado.

V. Implementación de modelo para GPU

Esta sección describe la implementación el modelo propuesto en la sección IV-B para las instancias de Controller dedicadas GPU NVIDIA, compatibles con la tecnología CUDA. Además, se describen las políticas seguidas para los puntos de sincronización en la dependencia de datos entre las operaciones de comunicación y los lanzamientos de kernels. También se diferencian los estados posibles de una instancia de Controller en la ejecución de un programa con operaciones asíncronas.

La política se activa en tiempo de ejecución mediante la variable de entorno `CAL_CNTRL_ASYNC_MODE`.

A. Concurrencia con CUDA

La librería Controller hace uso de la tecnología CUDA para el código específico que va a ser ejecutado sobre aceleradoras GPU de NVIDIA. CUDA ofrece una funcionalidad llamada *stream*, que se describe en [4] como una cola FIFO de operaciones GPU que son ejecutados en un orden específico. Algunos modelos de aceleradoras de NVIDIA soportan el uso de varios streams en un dispositivo. Las aceleradoras GPU de NVIDIA tienen un stream de defecto por el que se realizan todas las operaciones realizadas mediante la librería CUDA. La librería de Controller no utiliza este stream con la intención de tener un mayor control sobre la ejecución de cada tarea, así cada instancia de Controller tiene un stream propio.

En la arquitectura propuesta del Controller, la cola de tareas preparadas (parte derecha de la figura 2) se implementa mediante el stream de la instancia, al que se asignarán las transferencias de datos y los lanzamientos de kernels. Para permitir el solapamiento de comunicaciones y computación, estas operaciones deben producirse por diferentes streams del dispositivo. Por ello, en el modelo asíncrono del Controller, se le asigna un stream propio a cada estructura de datos al asociarla con un stream. Por este stream de la estructura se ejecutarán sus tareas de transferencia. Es necesario, liberar posteriormente este stream al desligar la estructura de la instancia de Controller (operación de detach).

La propuesta del Controller utiliza otra funcionalidad de CUDA para los procedimientos que deban hacerse tras la ejecución de una tarea de comunicación o de lanzamiento de kernel, las funciones callback. Estas funciones son ejecutadas por *el hilo del driver* CUDA asociado al dispositivo. Este hilo se ejecuta en la máquina anfitriona y es independiente de la arquitectura del Controller. Una función de callback debe ser asignada a un stream después de la incorporación de una operación de CUDA. Esta función será ejecutada cuando dicha operación se haya completado. El hilo del driver tiene una cola de callbacks pendientes a la que se irán incluyendo aquellas que estén preparadas para ser ejecutadas; este hilo las evaluará de forma sincrónica mediante política FIFO.

B. Modificaciones sobe estructura HitTile

Un HitTile, como se describió en la sección III, almacena meta-datos de una estructura de datos HitTile. La información que esta estructura almacena sobre el Controller es: `refCntrl`, puntero a la instancia de Controller a la que está asociada; `devData`, puntero a la dirección de memoria de la imagen de la estructura en el dispositivo; `inCntrlType`, identificador de tipo de estructura que

guarda el HitTile (aquellas variables que solo se encuentran en el dispositivo toman el nombre de *internal*); `transferred`, flag que determina si una estructura ha sido transferida al dispositivo asociado a la instancia de Controller; `recover`, flag que determina si la información de una estructura ha sido recuperada por la máquina anfitriona.

El modelo de sincronización implementado, que evita los problemas de concurrencia entre los hilos involucrados en la ejecución de Controller, se apoya en los siguientes miembros añadidos al metadata de la estructura: `handler`, referencia al stream asociado a la estructura; `hasPendingMoveTo`, flag que determina si la estructura de datos tiene pendiente una transferencia al dispositivo; `hasPendingMoveFrom`, flag que determina si la estructura de datos tiene pendiente una transferencia desde el dispositivo; `isCntrlBlocked`, flag que determina si el hilo de la instancia de Controller está bloqueada por la estructura de datos debido a una comunicación; `cntrlKernelCount`, contador de kernels que tienen como parámetro de entrada a la estructura de datos.

C. Definición de operaciones de comunicación

Este apartado describe las implementaciones de las operaciones de comunicación introducidas en la sección IV-A.

Las transferencias de las estructuras de datos se distinguen según dos criterios: a) si es necesario o no, reservar/liberar recursos en la máquina anfitriona o en el dispositivo; b) dirección de la transferencia de la comunicación, al/del dispositivo asociado a la instancia del Controller. El criterio a) distingue las operaciones attach y detach de dos nuevas operaciones de actualización de las estructuras de datos entre la memoria de la máquina anfitriona y la del dispositivo: (1) *MoveTo* (transferencia de datos al dispositivo), y (2) *MoveFrom* (transferencia de datos desde el dispositivo). Estas nuevas operaciones implican únicamente transferencias de datos. Según el criterio b), estas transferencias están implícitas en las dos operaciones del modelo anterior de Controller: (1) movimiento de datos al dispositivo después de la reserva de recursos (attach), y (2) recuperación de la estructura desde el dispositivo antes de la liberación de memoria en este (detach).

En la implementación del modelo, se diferencian tres tipos de funciones callbacks correspondientes a las operaciones de comunicación y de ejecución de kernels: (1) tras una comunicación MoveTo y (2) tras una comunicación MoveFrom. En ambas operaciones, se indica a la estructura de datos HitTile que la operación ha sido completada. (3) Tras un lanzamiento de kernel. Se le indica a todos los HitTile involucrados en ese lanzamiento que el kernel ha finalizado, actualizando el contador de kernels pendientes de HitTile.

D. Políticas de dependencia de datos entre comunicación y computación

A continuación se describen los diferentes escenarios con posibles problemas de concurrencia:

A) *Dependencia entre kernel y comunicación MoveTo:* Un kernel no puede iniciarse hasta que todos los HitTile involucrados en su ejecución, como parámetros de entrada, hayan sido transferidos al dispositivo. Es decir, la comunicación MoveTo que implica cada una de estos parámetros tiene que haberse completado, actualizando el campo `hasPendingMoveTo`.

B) *Dependencia entre comunicación MoveFrom y kernel:* Todas las operaciones de transferencia desde el dispositivo a la máquina anfitriona de un HitTile, tienen que esperar hasta que los kernels pendientes de ejecución que utilicen ese HitTile como parámetro de salida finalicen. En ese momento, el campo `cntrlKernelCount` contiene el valor 0.

El modelo propuesto contiene tres hilos de ejecución. Todos y cada uno de estos hilos se ejecutan en la máquina anfitriona:

Hilo principal: hilo de ejecución del código desarrollado por el usuario. Este hilo lleva el control de la aplicación, con la gestión de las variables e instancias de Controller y el flujo de ejecución. Este hilo también es el encargado de crear y añadir a las tareas a las instancias de Controller mediante las llamadas a funciones de la librería.

Hilo de Controller: hilo OpenMP que gestiona la ejecución de una instancia de Controller. Este hilo es el que extrae y evalúa las tareas de la cola de una instancia de Controller.

Hilo de driver de CUDA: hilo independiente del modelo de Controller. Este hilo es añadido por una funcionalidad CUDA y es el encargado de las ejecuciones de las funciones callback asociadas a un dispositivo.

Se ha elegido los semáforos POSIX [17] para los procedimientos de sincronización del modelo propuesto. Los miembros añadidos a la estructura de Controller para la abstracción sobre dispositivos GPU de NVIDIA (estructura `CALCntrlGPU`) son: `semHost`, semáforo dedicado para el control de sincronización con el hilo principal de la aplicación paralela (parte izquierda de la figura 2); `semCntrl`, semáforo dedicado para el control de sincronización con el hilo de la instancia de Controller (parte central de la figura 2). Se utilizan dos semáforos distintos ya que el modelo propuesto restringe las sincronizaciones entre los hilos participantes (ver figura 2), en las siguientes dos interacciones entre estos hilos: (1) las sincronizaciones que involucren al hilo principal de la ejecución (llamadas a la función `sem_wait` de la librería `semaphore.h`), se verán resueltas únicamente por llamadas a funciones de Controller desde el hilo de la instancia de Contro-

ller (llamadas a la función `sem_post` de la librería `semaphore.h`); (2) las sincronizaciones que involucren al de la instancia de Controller, se verán resueltas únicamente por llamadas de `sem_post` desde el hilo CUDA del driver del dispositivo.

Se añade una función de sincronización a la librería de Controller como método de espera por una estructura de datos en particular, la función `WaitTile`. Esta función ofrece la capacidad de parar la ejecución del programa principal hasta que las transferencias de la estructura, anteriores al punto de espera, finalicen. Es necesario realizar una espera para garantizar que la comunicación de la estructura de datos se ha finalizado y asegurar la consistencia de la estructura de datos en el espacio de memoria principal. En el modelo propuesto, el usuario es el responsable de establecer un punto de espera dentro del código de la aplicación.

D.1 Pseudocódigo de políticas de dependencias

Este apartado describe los procedimientos que realiza la librería de Controller para evitar los problemas de concurrencia descritos en anteriormente.

La política ideada para resolver dependencia A) (comunicación MoveTo – Kernel) se describe a continuación. Esta sincronización involucra los hilos de la instancia de Controller y el del driver de CUDA. Las operaciones envueltas en esta dependencia son: (1) *tarea de comunicación MoveTo* y (2) *lanzamiento de kernel*, ejecutados por el hilo de la instancia de Controller; (3) *callback tras una comunicación MoveTo*, ejecutado por el hilo del driver de CUDA. El algoritmo 1 explica las rutinas que se llevan a cabo, respectivamente.

Para resolver la dependencia B) (Kernel – comunicación MoveFrom). Los hilos involucrados en el procedimientos son los mismos que en el caso anterior: el hilo de la instancia de Controller y el hilo del driver de CUDA. Las operaciones que se ven afectadas por el punto de sincronización son: (1) *lanzamiento de kernel*, (2) y *tarea de comunicación MoveFrom* ejecutados por el hilo de la instancia de Controller; (3) *callback de lanzamiento de kernel*, ejecutado por el hilo del driver de CUDA. El algoritmo 2 explica las rutinas que se llevan a cabo para resolver esta dependencia.

Finalmente, la estrategia para establecer un punto de sincronización, que bloquee el hilo principal de forma explícita, compromete a los tres hilos de la ejecución del programa paralelo (ver figura 2). Las operaciones que se ven afectadas por el punto de sincronización son: (1) *inclusión de tarea de WaitTile*, ejecutada por el hilo principal; (2) *tarea de MoveFrom* y (3) *tarea de WaitTile*, ejecutadas por el hilo de la instancia de Controller; (4) *callback de comunicación MoveFrom*, ejecutado por el hilo del driver de CUDA. El algoritmo 3 explica las rutinas que se llevan a cabo, respectivamente.

Todas las operaciones de lectura y escritura de los miembros de una estructura de datos se realizan mediante operaciones atómicas. El uso de operaciones atómicas se debe a la intención de mantener consistencia de memoria en las operaciones de lectura y escritura sobre las estructuras de datos compartidas por los hilos en ejecución.

Algoritmo 1: Política de dependencia comunicación MoveTo – Kernel

```
Algoritmo MoveTo
    Entrada:
    cntrl: reference of Controller instance;
    tile: reference of data structure;
    inicio
        tile.hasPendingMoveTo → true;
        // Ejecución de transferencia
           MoveTo
    fin
fin
Algoritmo KernelLaunch
    Entrada:
    cntrl: reference of Controller instance;
    kernel: reference of task of kernel launch;
    inicio
        para cada tile ∈ kernel.parameters
         hacer
            si tile.role = input entonces
                tile.isCntrlBlocked ← true;
                si tile.hasPendingMoveTo =
                 true entonces
                    wait(cntrl.semCntrl);
                en otro caso
                    tile.isCntrlBlocked ←
                     false;
                fin
            fin
        fin
        // Ejecución del lanzamiento del
           kernel
    fin
fin
Algoritmo CallbackMoveTo
    Entrada:
    tile: reference of data structure;
    inicio
        tile.hasPendingMoveTo ← false;
        si tile.isCntrlBlocked = true
         entonces
            tile.isCntrlBlocked ← false;
            post(tile.refCntrl.semCntrl);
        fin
    fin
fin
```

VI. Estudio experimental

Esta sección muestra los experimentos que evalúan las posibles ventajas y las limitaciones de la integración y el solapamiento de comunicaciones y computación en la librería Controller. Esta sección incluye: A) una descripción de los casos de estudio elegidos, B) el estudio de rendimiento sobre nuestra propues-

Algoritmo 2: Política de dependencia Kernel – comunicación MoveFrom

```
Algoritmo KernelLaunch
    Entrada:
    cntrl: reference of Controller instance;
    kernel: reference of task of kernel launch;
    inicio
        // Política de dependencia
           comunicación MoveTo -- Kernel
        para cada tile ∈ kernel.parameters
         hacer
            si tile.role = output entonces
                tile.cntrlKernelCount ←
                 tile.cntrlKernelCount + 1;
            fin
        fin
        // Ejecución del lanzamiento del
           kernel
    fin
fin
Algoritmo MoveFrom
    Entrada:
    cntrl: reference of Controller instance;
    tile: reference of data structure;
    inicio
        tile.isCntrlBlocked ← true;
        si tile.cntrlKernelCount > 0
         entonces
            wait(cntrl.semCntrl);
        en otro caso
            tile.isCntrlBlocked ← false;
        fin
        // Ejecución de transferencia
           MoveFrom
    fin
fin
Algoritmo CallbackKernelLaunch
    Entrada:
    kernel: reference of task of kernel launch;
    inicio
        para cada tileinkernel.parameters
         hacer
            si tile.role = output entonces
                tile.cntrlKernelCount ←
                 tile.cntrlKernelCount − 1;
                si tile.cntrlKernelCount =
                 0 ∧ tile.isCntrlBlocked = true
                 entonces
                    tile.isCntrlBlocked ←
                     false;
                    post(tile.refCntrl.semCntrl);
                fin
            fin
        fin
    fin
fin
```

Algoritmo 3: Operación de sincronización con una estructura de datos: Creación de tarea de WaitTile

```
Algoritmo CreateTaskWaitTile
    Entrada:
    cntrl: reference of Controller instance;
    tile: reference of data structure;
    Datos:
    task: initialize task of synchronization
    with data structure;
    inicio
        cntrl.listTask ←
         cntrl.listTask ⋃ {task};
        wait(cntrl.semHost);
    fin
fin
Algoritmo MoveFrom
    Entrada:
    cntrl: reference of Controller instance;
    tile: reference of data structure;
    inicio
        // Política de dependencia
           comunicación Kernel --
           MoveFrom
        tile.hasPendingMoveFrom ← true;
        // Ejecución de transferencia
           MoveFrom
    fin
fin
Algoritmo WaitTile
    Entrada:
    cntrl: reference of Controller instance;
    tile: reference of data structure;
    inicio
        tile.isCntrlBlocked ← true;
        si tile.hasPendingMoveFrom = true
         entonces
            wait(cntrl.semCntrl);
        en otro caso
            tile.isCntrlBlocked → false;
        fin
        post(cntrl.semHost);
    fin
fin
Algoritmo CallbackMoveFrom
    Entrada:
    tile: reference of data structure;
    inicio
        tile.hasPendingMoveFrom ← false;
        si tile.isCntrlBlocked = true
         entonces
            tile.isCntrlBlocked ← false;
            post(tile.refCntrl.semCntrl);
        fin
    fin
fin
```

ta.

A. Casos de estudio

Hemos seleccionado dos casos de estudio diferentes para validar el modelo propuesto. Todas ellos trabajan con estructuras de datos con valores de tipo *float*.

Jacobi bi-dimensional: Este caso de estudio computa la estabilidad puntual de la Ecuación Diferencial Parcial de Poisson (PDE) de la difusión de calor. Utiliza el método iterativo de Jacobi para un espacio discreto bi-dimensional. Este programa stencil de cuatro punto ejecuta un número de iteraciones fijo. En cada iteración se calcula un nuevo valor por cada celda de la matriz, utilizando la información de los cuatro vecinos superior, inferior, izquierdo y derecho. El resultado después de cada iteración se transfiere a la máquina anfitriona.

Potencia de una matriz: Este caso de estudio calcula $C = A^n$. Se elevar una matriz cuadrada A por una potencia n. Hemos escogido una implementación basada en un bucle que computa la expresión recursiva: $C_i = C_{i-1} \times A$ para todo $k \in 1, 2, \ldots, n$ donde $C_0 = A$. El kernel que multiplica ha sido escogido del ejemplo optimizado contenido en los ejemplos de CUDA ofrecidos por NVIDIA. Esta implementación utiliza memoria compartida entre hilos en el dispositivo y desenrollado de bucles para obtener mejoras en el uso de los recursos de la GPU.

B. Estudio de rendimiento

Se han desarrollado versiones síncronas y asíncronas con la tecnología CUDA para los casos de estudio escogidos. Denominamos como versiones base a estas soluciones para la comparación de rendimiento. El código de Controller tiene una única implementación, y la política de transferencias asíncronas se activa en tiempo de ejecución. Los objetivos del estudio de rendimiento son: (1) comparar entre implementaciones programadas con la librería de CUDA y otra utilizando la librería de Controller; (2) comparar entre versiones síncorna y asíncrona de Controller, utilizando el modelo propuesto.

La ejecución de los códigos de prueba se ha llevado a cabo en una máquina con las siguientes especificaciones: Intel Xeon E5-2690 v3 @1.9 GHz, una memoria RAM principal 64 GB GDDR3, y cuatro aceleradoras GPU, NVIDIA's GTX Titan BLACK, with 2880 cores @ 980 MHz y 6 GB de memoria.

Las figuras 3 y 4 muestran los resultados de rendimiento obtenidos para los casos de estudio. El eje ordenadas representa el tiempo total de ejecución (incluyendo los tiempos computación y comunicación) en escala logarítmica decimal. Los diferentes tamaños de las matrices involucradas en los casos de estudio se representan en el eje de abcisas. Para el caso de estudio de la potencia de una matriz, el eje de abcisas tiene escala logarítmica en base 2. En las gráficas, se añaden etiquetas por cada tamaño dado el porcentaje de mejora entre las ejecuciones de la version de Controller (*Controller Asynchronous* vs *Controller Synchronous*).

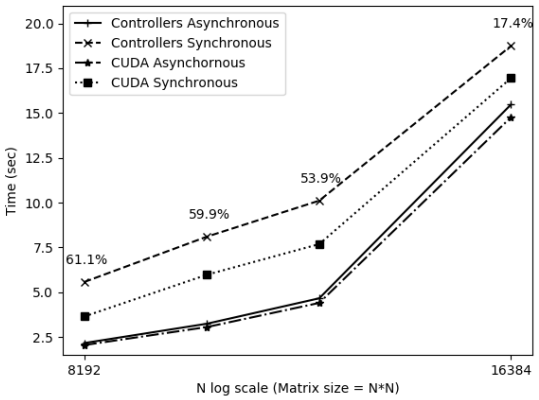

Fig. 3
Gráficas comparativas de rendimiento – Jacobi bi-dimensional con 100 iteraciones

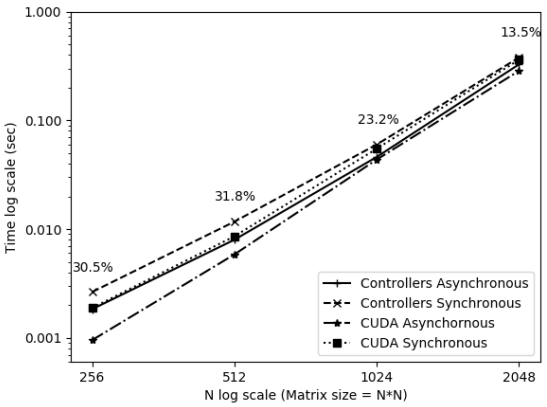

Fig. 4
Gráficas comparativas de rendimiento – Elevar una matriz a la décima potencia: $C = A^{10}$

En la evolución del porcentaje de mejora de rendimiento mostrados en las figuras 3 y 4, se observan las mejoras de tiempo debidas al solapamiento de tareas de comunicación y computación. En el caso de estudio de Jacobi bi-dimensional con 100 iteraciones, el porcentaje de rendimiento entra los modelos síncrono y asíncrono de Controller se encuentran entre 17,4 % y 61,1 %. En el caso de estudio de la potencia de una matriz elevada a la 10, los porcentajes de mejora están entre 13,5 % y 31,8 %. Ambos casos de estudio muestran una simititud en la evolución del rendimiento de las soluciones síncronas y las soluciones asíncronas entre sí. Las versiones de Controllers muestran una pequeña penalización temporal, respecto a las soluciones en CUDA, debidas al coste extra de las llamadas internas de la librería.

VII. Conclusiones

Este trabajo presenta una propuesta de modelo para solapar tareas de comunicación y computación en dispositivos GPU. Se ha establecido una política de dependencias de datos para que la ejecución solapada de ambos tipos de tareas no produzca inconsistencias. El usuario no necesita recurrir a nuevas funciones sobre la versión anterior de Controllers, para explotar las funcionalidades presentadas en este trabajo. El uso del modelo propuesto ha obtenido un porcentaje hasta del 61,10 % de mejora frente a la versión secuencial de Controller en la experimentación realizada.

El trabajo futuro incluye la extensión del modelo para otros tipos de aceleradores, como los procesadores XeonPhi o FPGA, o la inclusión de otras políticas más sofisticados para gestión de colas.

Referencias

[1] J. Dongarra, H. Meuer, and E. Strohmaier, "Top500 Supercomputer Sites," Tech. Rep., University of Tennessee, Knoxville, TN, USA, 1998.

[2] Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos, "MPI-aware Compiler Optimizations for Improving Communication-computation Overlap," in *Proceedings of the 23rd International Conference on Supercomputing*, New York, NY, USA, 2009, ICS '09, pp. 316–325, ACM.

[3] David B. Kirk and Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[4] Jason Sanders and Edward Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, 1st edition, 2010.

[5] John E. Stone, David Gohara, and Guochun Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010.

[6] Arturo Gonzalez-Escribano, Yuri Torres, Javier Fresno, and Diego R. Llanos, "An Extensible System for Multilevel Automatic Data Partition and Mapping," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1145–1154, May 2014.

[7] Ana Moreton–Fernandez, Hector Ortega–Arranz, and Arturo Gonzalez–Escribano, "Controllers: An abstraction to ease the use of hardware accelerators," *The International Journal of High Performance Computing Applications*, p. 109434201770296, May 2017.

[8] "Grupo Trasgo – Trasgo Research Group (Universidad de Valladolid) - Home Page," .

[9] M. Sourouri, J. Langguth, F. Spiga, S. B. Baden, and X. Cai, "CPU+GPU Programming of Stencil Computations for Resource-Efficient Use of GPU Clusters," in *2015 IEEE 18th International Conference on Computational Science and Engineering*, Oct. 2015, pp. 17–26.

[10] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers, "CIVL: the concurrency intermediate verification language," in *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2015, pp. 1–12.

[11] Leonardo Dagum and Ramesh Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[12] Message P Forum, "MPI: A Message-Passing Interface Standard," Tech. Rep., University of Tennessee, Knoxville, TN, USA, 1994.

[13] Antonio Vilches, Rafael Asenjo, Angeles Navarro, Francisco Corbera, Rubén Gran, and María Garzarán, "Adaptive Partitioning for Irregular Applications on Heterogeneous CPU-GPU Chips," *Procedia Computer Science*, vol. 51, no. Supplement C, pp. 140 – 149, 2015.

[14] Borja Pérez, José Luis Bosque, and Ramón Beivide, "Simplifying Programming and Load Balancing of Data Parallel Applications on Heterogeneous Systems," in *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, New York, NY, USA, 2016, GPGPU '16, pp. 42–51, ACM.

[15] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang, "GDM: Device Memory Management for Gpgpu Computing," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, 2014, SIGMETRICS '14, pp. 533–545, ACM.

[16] J. Lee, M. Samadi, and S. Mahlke, "VAST: The illusion of a large memory space for GPUs," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug. 2014, pp. 443–454.

[17] "IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications, Issue 7," Dec. 2008.