Transferencias de datos asíncronas y transparentes en plataformas heterogéneas

Victor Lara-Mongil[1], Ismael Taboada-Rodero[1], Eduardo Rodriguez-Gutiez[2], Yuri Torres[2], Arturo Gonzalez-Escribano[2] y Diego R. Llanos[2]

Resumen— **Los coprocesadores de alto rendimiento, como las Unidades de Procesamiento Gráfico (GPUs), presentan un ratio alto entre rendimiento y coste junto con un bajo consumo de energía. Por ello, los sistemas heterogéneos que los incluyen han experimentado un crecimiento significativo. Sin embargo, la programación de estos dispositivos sigue suponiendo un reto. Uno de los problemas está relacionado con la gestión de la memoria. Estos dispositivos tiene su propio espacio de memoria y es necesario realizar costosas transferencias de datos entre la máquina anfitriona y el dispositivo.**

En este trabajo proponemos una novedosa solución en tiempo de ejecución que analiza las dependencias de las diferentes transferencia de datos, ejecución de kernels y operaciones de host, solapándolas, en la medida de lo posible, de forma automática. Esta solución puede ocultar las latencias de forma transparente, mejorando significativamente el rendimiento de la aplicación. La técnica propuesta está implementada en el modelo de programación de Controllers para plataformas heterogéneas.

Presentamos un estudio experimental que compara programas desarrollados utilizando nuestra solución con programas desarrollados con CUDA y OpenCL. Las versiones implementadas consideran tanto transferencias síncronas como asíncronas. El estudio muestra que la abstracción propuesta introduce un sobrecoste despreciable, mientras que mejora el tiempo de ejecución y reduce el esfuerzo de desarrollo del programa, evitando el uso explícito de mecanismo de asincronía. Los resultados ofrecen hasta un 44.6 % de reducción del tiempo de ejecución de una aplicación real de retransmisión de vídeo, debido al solapamiento de las transferencias de datos y la ejecución de los kernels.

Palabras clave— **Computación heterogénea, sistemas en tiempo de ejecución, ocultación de latencia, ejecución asíncrona.**

I. Introducción

LOS coprocesadores de alto rendimiento, como las Unidades de Procesamiento Gráfico (GPUs), han sido ampliamente adoptadas en los sistemas modernos para la computación de propósito general. Esto se refleja en la configuración de muchos supercomputadores en las posiciones más altas del ranking TOP500 [1]. Sin embargo, el esfuerzo necesario para la explotación eficiente de estos modelos de programación para estos dispositivos, como OpenCL, CUDA, o OpenMP, sigue siendo un desafío.

Muchos modelos de programación para plataformas heterogéneas y coprocesadores usan el concepto de *kernel*. Esta es una unidad de código que se compila para un dispositivo de procesamiento específico, cuya ejecución puede ser solicitada por el programa principal. Los procesadores multi- o many-core explotan los recursos hardware subyacentes para paralelizar la ejecución de los kernel, pero estos deben ser lanzados, ejecutados y sincronizados como uno solo. Las transferencias de datos entre las jerarquías de memorias de la máquina anfitriona, en adelante máquina *host*, y el coprocesador, se expresan en un grado grueso. La petición de estas transferencias normalmente implican el movimiento de estructuras de datos completas que minimicen los costes asociados a las operaciones de transferencia individuales. Los programas están estructurados como una secuencia de llamadas a interfaces de programación específicas, mezclando transferencias de datos y lanzamiento de kernels con código ejecutado en el host. Cuando estas operaciones se ejecutan en orden, la semántica se mantiene sin alteraciones. En cambio, las operaciones de transferencia de datos y los kernels son costosas, y su ejeución síncrona produce retrasos importantes en el camino crítico del programa (ver ejemplo de la figura 1).

Para solventar este problema, muchos coprocesadores e interfaces de programación permiten el solapamiento de computación y transferencias de memorias mediante el uso de operaciones asíncronas. La solución a este conocido problema de solapamiento de transferencias y computación [2] supone un desarrollo tedioso y propenso a errores. Ello implica el uso de conceptos complejos y funcionalidades específicas soportadas por la interfaz de programación (como *streams* o *events*), la creación de funciones *callback*, y el uso de mecanismos de sincronización. A su vez, este desarrollo requiere el análisis del programa para detectar aquellas situaciones donde se pueden explotar las operaciones asíncronas sin afectar al comportamiento del programa.

En este trabajo proponemos una solución para aprovechar e introducir ejecuciones asíncronas en secuencias genéricas de transferencia de datos, ejecuciones de código host y computación de kernels, en diferentes tipos de coprocesadores. Nuestra técnica analiza en tiempo de ejecución la secuencia de operaciones solapando, de forma automática, la transferencia de datos y la computación siempre que sea posible. La solución está implementada como una política de gestión de cola de trabajos en el modelo de programación de Controllers para plataformas heterogéneas [3], [4]. La interfaz y funcionalidades del Controller han sido rediseñadas para proveer una librería genérica para el uso de operaciones básicas que han sido consideradas en esta propuesta. Esta nueva

[1]Dpto. de Informática, Univ. de Valladolid, e-mail: {victor.lara|ismael.jose.taboada}@uva.es.
[2]Dpto. de Informática, Univ. de Valladolid, e-mail: {eduardo|yuri.torres|arturo|diego}@infor.uva.es.

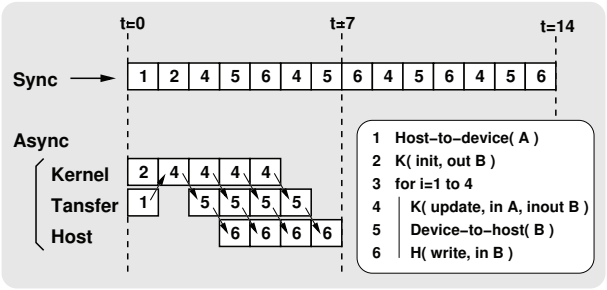

Fig. 1. Ejemplo de modos de ejecución síncrona y asíncrona de una secuencia de operaciones. El modelo asíncrono permite el solapamiento de las transferencias de datos con la ejecución de kernels en el dispositivo K, y con la ejecución de funciones en el host H. las flechas indican las dependencias entre las operaciones en diferentes canales asíncronos. La aceleración obtenida con la ejecución asíncrona frente a la síncrona es de $S = 2$.

interfaz de programación (API) no expone los detalles o mecanismos específicos para el uso de operaciones asíncronas. También, hemos diseñado, implementado y probado dos backends mediante el uso de CUDA para GPUs de Nvidia, y de OpenCL para GPUs de AMD a mayores. Los modelos de ejecución síncrona y asíncrona se pueden seleccionar en tiempo de ejecución, y el solapamiento de transferencias de datos y computación es totalmente transparente.

Además, presentamos un estudio experimental con un caso que muestra la flexibilidad y sencillez del uso de nuestra propuesta. Esta experimentación refleja como las nuevas técnicas mejoran, de forma transparente, el rendimiento de las aplicaciones siempre y cuando sea posible el solapamiento. Comparamos el rendimiento y los códigos de aplicaciones de referencia programadas y ajustadas en CUDA y OpenCL. Nuestra aproximación muestra como se reduce el esfuerzo de desarrollo necesario para desarrollar un código asíncrono, consiguiendo un mayor rendimiento y portabilidad.

El resto del trabajo se organiza en las siguientes secciones. La sección II presenta el trabajo relacionado. La sección III introduce los conceptos previos sobre el modelo anterior de Controller. En la sección IV se describe el nuevo modelo asíncrono de Controller y se detalla la implementación con CUDA para su uso en dispositivos Nvidia. La sección V muestra la experimetación realizada para validar la implementación del modelo propuesto en términos de rendimiento y métricas de desarrollo de software. Por último, la sección VI concluye el trabajo, exponiendo los resultados obtenidos e indicando posibles trabajos futuros.

II. Trabajo relacionado

Existen tres principales aproximaciones para la coordinación paralela de dispositivos heterogéneos. La primera aproximación consiste en el desarrollo de soluciones a medida usando modelos paralelos nativos, como CUDA. Algunos frameworks para entornos GPGPU [5], [6] mezclan estos modelos nativos con el uso de otros modelos ajenos a los coprocesadores como OpenMP y MPI. Con esta aproximación es posible controlar de forma eficiente los recursos del hardware así como su configuración, pero el desarrollador debe tener un conocimiento avanzado de las particularidades del hardware y los diferentes modelos de programación.

Otras herramientas y librerías proponen enfoques más abstractos orientados a la paralelización automática de bucles en dispositivos heterogéneos, tales como LogFitc [7], or Maat [8]. Éstas dividen, de forma transparente, las iteraciones de un bucle en tareas mediante el uso de técnicas de transferencias asíncronas, y ejecutan las tareas en los dispositivos utilizando diferentes políticas de gestión de procesos o *scheduling*. Sin embargo, estos trabajos no tiene una coordinación entre grafos de tareas genéricas y dependencias de datos, así como las que pueden generarse en secciones con bucles anidados.

Algunos frameworks y herramientas de programación, como dCUDA [9], G-Charm [10], o los nuevos Executors propuestos para C++ [11], introducen comunicaciones asíncronas entre el host y los dispositivos GPU. Éstos solapan las tareas de comunicación y computación. En cambio, el programador necesita especificar de forma manual los mecanismo de sincronización o implementar funciones de callback para los kernels. Por lo tanto, el uso de las comunicaciones asíncronas no es transparente. Asimismo, dCUDA, Groute, y BlasX no resuelven de forma automática las dependencias de datos.

Otros sistemas de programción, tales como Gdev [12], y VAST [13], proponen abstracciones para la gestión de espacios virtuales de memoria para dispositivos GPU. Estos sistemas simplifican el acceso transversal entre el host y las GPUs. Utilizan diferentes políticas pasivas (*lazy*) o activas (*eager*) para el movimiento de datos y paginación, de forma similar a los mecanismos existentes para memoria virtual. No obstante, estos sistemas no introducen formas para hacer transparente el solapamiento de tas transferencias de estructuras de datos completas y la operaciones de ejecución de kernels tomando en cuenta las dependencias a grano grueso.

Ninguno de los enfoques mencionados resuelve el problema del solapamiento transparente de computación y transferencias de datos, para coprocesadores heterogéneos con jerarquías de memorias propias.

III. Modelo de Controllers

En esta sección se resume las características del modelo original de Controllers para plataformas heterogéneas [3]. A continuación se describen los conceptos básicos relacionados con la programación mediante el uso de este modelo, utilizando fragmentos de código ilustrativos. Se añade un ejemplo de una aplicación que consiste en una programa iterativo que aplica el filtro de Sobel a los fotogramas (*frames*) de un flujo de vídeo.

La operación de Sobel [14] procesa imágenes en escala de grises para detectar bordes. Esta aplica dos operaciones stencil a una imagen de entrada obte-

niéndo las derivadas de los colores en los ejes X e Y. La magnitud del gradiente que se computa en cada celda como la distancia Euclidiana con las correspondientes celdas restante de la matrices que se obtienen como salida del filtro. La figura 2 muestra el código con Controllers de un programa de un filtro de Sobel que envía y recibe un fotograma por iteración hacia y desde el dispositivo. En este ejemplo, el nivel de cada pixel está representado por un número real de precisión sencilla (*float*). El código de un kernel utilizado en este programa puede verse en la figura 3.

```
1  CTRL_KERNEL_PROTO( saxpy_sqrt, 1, GENERIC,
2          3,
3          IVAL, int, alpha,
4          IN, HitTile_float, matrix_x,
5          IO, HitTile_float, matrix_y );
6
7  CTRL_KERNEL( saxpy_sqrt, GENERIC,
8          int alpha,
9          KHitTile_float matrix_x,
10         KHitTile_float matrix_y,
11 {
12     const int row = threadId.x;
13     const int col = threadId.y;
14     hit(matrix_y, row, col) = sqrt(
15          alpha * hit(matrix_x, row, col) +
16                  hit(matrix_y, row, col) );
17 } );
```

Fig. 3. Ejemplo de prototipo e implementación de un kernel usando la libería Controller.

A. El modelo

El modelo original de Controller propone un instancia abstracta que coordina las actividades de ejecución de kernel y manejo de memoria en una aceleradora o coprocesador, o en un conjunto de cores de CPU. En la construcción de una instancia de Controller se le asocia a un dispositivo. Los métodos que dispone esta instancia permiten (1) asociar/desasociar estructura de datos del host al dispositivo, y (2) lanza ejecuciones de kernels usando estas estructuras de datos como parámetros.

Las operaciones de lanzamientos de kernel (*kernel launch*) introducen peticiones de ejecución dentro de una cola interna de la instancia del Controller. Los parámetros que se utilicen como entrada de la función o las estructuras de datos deben estar previamente asociadas a la instancia. El lanzamiento de un kernel también recibe un parámetro que indica el número de hilos lógicos con los que debe ejecutarse dicho kernel. El Controller maneja la granularidad necesaria, agrupando estos hilos en bloques o en tareas, para adaptarlos a las especificaciones del dispositivo. La operación de asociar (*attach*) bloquea la ejecución hasta que se ha asociado correctamente el espacio de memoria en el dispositivo y se transfieren los datos del host al dispositivo. La operación de desasociar (*detach*) recupera los datos desde el dispositivo al host y libera la memoria en el dispositivo. Esta última operación también es bloqueante, finaliza cuando todos los kernels que han sido encolados y hacen uso de los datos han terminado y los datos han sido movidos al host desde el dispositivo. Si no hay ningún kernel que tenga la estuctura de datos como salida de la función, la operación de detach simplemente desaloja la memoria del dispositivo.

B. Representación de estructura de datos: Hitmap

Hitmap [15] es una librería protable que es utilizada en el modelo de Controller para ofrecer una interfaz común para la gestión de datos en código de host y los kernels ejecutados en el dispositivo. Una estructura de *HitTile* es un *fat pointer*, un manejador para almacenar medatados además del puntero al espacio de memoria. El modelo de Controllers extiende la estructura de HitTile para almacenar metadatos relacionados con el uso de un instancia de HitTile (en adelante *tile*) en el dispositivo, incluyendo el puntero al espacio de memoria en el dispositivo. Entre las líneas 7–13 en la figura 2 muestra como se declaran tiles con y sin memoria en el host. La línea 17 refleja como se asocian los tiles a una instancia de Controller. Los tiles sin memoria en el host se utilizan como referencias para los buffers internos que están en el dispositivo. Las operaciones de asociación de tiles a una instancia de Controller no implican movimientos de memoria, sólo el alojamiento y desalojo de memoria. La función *hit* se usa en los códigos de host y de kernel para cceder a los elementos de un tile. Esta permite un acceso portable, con un orden *row-major*, en cualquier dispositivo. Ver las líneas 14–16 en la figura 3.

C. Kernels y su implementación

En el modelo de Controller, un kernel se declara mediante dos primitivas. La primera es `CTRL_KERNEL_PROTO`, que declara el prototipo de todas las implementaciones del kernel. Ver las líneas 1–5 de la figura 3. Después del nombre del kernel, declaramos en número de las diferentes implementaciones disponibles para el proagrama. También se indica para que backends o dispositivos están dedicadas las implementaciones. Esta información es usada por el Controller para localizar en tiempo de ejecución la mejor implementación dispoible del kernel para un dispositivo en concreto. En el ejemplo hay únicamente una implementación declarada. La palabra clave `GENERIC` indica que puede ser utilizada por cualquier backend. Otras parablas clave estan asociadas a tipos específicos de backend o dispositivos. El resto del prototipo describe los parámetros del kernel, incluyendo los roles de entrada/salida, tipos y nombres. `INVAL` indica un parámetro por valor. `IN`, `OUT`, o `IO` (in/out) indica los roles que puede tomar un HitTile que se pasa por referencia.

Cada implementación de kernel se declara usando la primitiva `CTRL_KERNEL`. Ver líneas 7–17 en la figura 3. Los primero parámetros determinan el nombre, y el tipo de la implementación. Después de la declaración de los parámetros del kernel, se incluye el código de la impelemntación. El kernel del ejemplo computa la operación *saxpy*, fusiona la aplicación de las raíces cuadradas en cada uno de los elementos

```
1  int main(int argc, char *argv[]){
2      ...
3      //Controller creation
4      Ctrl ctrl = Ctrl_Create( CTRL_TYPE_CUDA, DEVICE_ID );
5
6      // Create data structures
7      HitTile_float GxC = hitTile( float, hitShape(3, 3) );
8      HitTile_float GxR = hitTile( float, hitShape(3, 3) );
9      HitTile_float imgOrig = hitTile( float, hitShape( rows, columns) );
10     HitTile_float imgGy = hitTile( float, hitShape( rows, columns) );
11     HitTile_float imgGx = hitTileNomem( float, hitShape( rows, columns) );
12     HitTile_float imgTmp1 = hitTileNomem( float, hitShape( rows, columns) );
13     HitTile_float imgTmp2 = hitTileNomem( float, hitShape( rows, columns) );
14
15     // Initialization in the host, and attachments
16     init( GxC, GxR );
17     Ctrl_Attach( GxC, GxR, imgGx, imgTmp1, imgTmp2 );
18
19     // Domain of logical threads for the device
20     Ctrl_Threads threads = Ctrl_Threads(rows, columns);
21
22     // Sobel filter: Main loop
23     // Clock: Synchronize and start measuring
24     for (int index = 0; index < ITERATIONS; index ++) {
25        getFrame( imgOrig ); // Host code to get a new frame
26        Ctrl_Attach( imgOrig ); // Attach the new image to the device
27        Ctrl_Launch(ctrl, convolutionRow, threads, imgTmp, imgTmp2, imgOrig, GxR, GxC);
28        Ctrl_Detach(ctrl, imgOrig ); // Detach to let the host reuse the tile
29        Ctrl_Launch(ctrl, convolutionColum, threads, imgGx, imgTmpGxC);
30        Ctrl_Attach(ctrl, imgGy ); // Attach the output/results buffer
31        Ctrl_Launch(ctrl, convolutionColum, threads, imgGy, imgTmpGxR);
32        Ctrl_Launch(ctrl, saxpySqrt, threads, imgGx, imgGy, 1.0);
33        Ctrl_Detach(ctrl, imgGy ); // Detach no retrieve the data
34        putFrame( imgGy ); // Host code
35     }
36     // Clock: Synchronize and stop measuring
37
38     // Free the device and host memory
39     Ctrl_Detach( ctrl, GxC, GxR, imgGx, imgTmp2, imgTmp2 );
40     Ctrl_Destroy( ctrl );
41     hit_tileFree( GxC, GxR, imgOrig, imgGy );
42     ...
43 } //main
```

Fig. 2. Fragmento de código del programa del filtro de Sobel aplicado sobre un flujo de vídeo. Está programado utilizando el modelo original de Controller.

del resultado. La implementación es una especificación genérica de paralelismo de datos a grano fino, que es adaptada por el Controller a la granularidad por tareas adecuada a los diferentes tipos de dispositivos, como GPUs, o Intel XeonPhi.

D. Codificación de programas con el modelo original de Controller

Un programa con Controllers se apoya en las siguientes pautas: (1) Creación de instancias de Controller, asociando cada una de ellas a un dispositivo que se quiera gestionar; (2) declaración, alojamiento, inicialización (si es necesario), y asociación de los tiles; (3) lanzamiento de una secuencia de kernels; (4) desasociación de los tiles para recuperar los datos en el host; y (5) la liberación de los recursos de Controller y las estructuras de datos.

La figura 2 muestra la implementación de un programa imperativo que procesa los fotogramas de un flujo de vídeo, aplicando el filtro de Sobel. Se crea una instancia de Controller en la línea 4 que utiliza el backend de CUDA, y se le asocia un dispositivo mediante su identificador. Las líneas 7–13 crean los tiles en el host, alojando la memoria en el host solo para uno de ellos que va a ser utilizado en el host. La línea 16 llama a la función de host para inicializar los tiles que contienen los pesos de las operaciones de convolución. Los tiles se asocian a la instancia del Controller en la línea 17, forzando su alojamiento en el dispositivo, y moviendo los datos de los tiles que tienen memoria en el host. La línea 20 crea una instancia que declara la lógica de indexado del espacio de los hilos a grano fino para los kernels. Entre las líneas 25 y 28 se ejecutan las operaciones de host que toman un nuevo fotograma, asocia el tile a la instancia del Controller, lanza el kernel que le procesa generando datos en los buffers internos del dispositivo y desasocia el tile para reutilizarlo en la siguiente iteración. Entre las líneas 29 y 33 se lanzan los kernels restantes. Es importante tener en cuenta que el tile que se utiliza para recuperar el resultado debe ser asociado antes de que un kernel lo utilice (línea 30), y que debe ser desasociado para recuperar estos datos (línea 33), antes de que los datos puedan ser utilizados en el host (línea 34).

El modelo original de Controllers se diseñó para aplicaciones que moviesen los datos de entrada al dispositivo al inicio del programa y retornase los resulta-

dos de vuelta al final de la computación. Por lo tanto, las operaciones para mover datos entre los espacios de memoria también gestionan el alojamiento de la memoria. Esto no es eficiente para programas con flujos de datos que necesitan transferencias de entrada y salida a los buffers en el dispositivo de forma repetida. Además, las semánticas de las operaciones de asociación de datos implican que se realizan de forma síncrona con la ejecución de los lanzamiento de los kernels antes/después, para preservar la equivalencia secuencial. Entonces, no permiten el solapamiento de computación y transferencias.

IV. El modelo asíncrono de Controller

En esta sección presentamos nuestra propuesta para el nuevo modelo asíncrono de Controllers tal que sea capa de explotar de forma eficiente y transparente las operaciones asíncronas y el solapamiento de computación y transferencia de memoria. Esta propuesta implica la redifinición de las operaciones soportadas por el Controller, la redifinición de su modelo de ejecución, y el soporte para las operaciones asíncronas en los backend implementados.

En el nuevo modelo asíncrono de Controller, tanto las tareas de kernel como las de host se identifican como unidades de computación que pueden usar y modificar los datos de un tile. Por lo tanto, las tareas de host deben ser declaradas y lanzadas de manera similar a los kernels, con prototipos que definan los roles de los parámetros, de tal manera que permita al model detectar las dependencias cuando un tile se utiliza como entrada o salida o ambas tanto en el host como en el dispositivo. Al igual que con los kernels, la ejecución de tareas de host es independiente de la ejecución del programa principal. Una vez lanzado, podrá ser ejecutado de forma concurrente a la ejecución principal por otro hilo.

En el nuevo modelo asíncrono de Controller, las operaciones de asociación de los tiles están obsoletas. Cada una es sustituida por dos operaciones, una de gestión de memoria (alojamiento y desalojo de memoria) y otra de transferencia de datos (desde y hacia el dispositivo). La transferencia de datos se puede aplicar sobre los tiles que han sido asociados previamente y tienen memoria alojada en ambos espacios de memoria del host y del dispositivo.

En esta propuesta, un *programa* para plataformas heterogéneas se ve como un código que coordina la ejecución de tareas de host y operaciones del coprocesador. Las partes del código host que procesan datos están recogidas dentro de tareas de host. El programa principal simplemente define la secuencia de peticiones que se incorporan a la cola del Controller. Esta cola está controlada por una política de ejecución. Hemos definidos dos políticas básicas: Una que ejecuta las operaciones en modo síncrono, como hacía el modelo clásico de Controlelr; y otra analiza las dependencias y ejecuta las operaciones en modo asíncrono, solapando la computación y las transferencias cuando sea posible.

A. Nuevos tipos de operaciones

Las transacciones entre el host y el dispositivo son descritas como una *secuencia de peticiones de operaciones* ($S = s_0, \ldots, s_n$) emitidas por el código principal que se ejecuta en el host. Una petición de operación (*request*) s_i puede ser uno de los siguientes tipos:

- **Allocate:** $Alloc(x)$ o $AllocDev(x)$. Petición para alojar una estructura de datos en la memoria del host o del dispositivo, o únicamente en el dispositivo, respectivamente.
- **Dealocate:** $Free(x)$. Petición para desalojar de la memoria para imágenes de una estructura de datos.
- **Host-to-Device:** $HTD(x)$. Petición para transferir los valores de la estructura de datos x desde el host al espacio de memoria del dispositivo.
- **Device-to-Host:** $DTH(x)$. Petición para transferir los valores de la estructura de datos x desde el dispositivo a la memoria del host.
- **Lanzamiento de kernel:** $K(f, In, Out)$. Petición para ejecutar un kernel en el dispositivo. Este lanzamiento recibe el nombre de la función f y dos conjuntos de estructuras de datos como parámetros. El conjunto In representa las estructuras de datos que son entradas. El otro conjunto Out contiene las referencias de las salidas. Una misma estructura de datos puede aparecer en ambos conjuntos, esto presenta el caso en el que el contenido de la estructura se lee y se escribe durante la ejecución del kernel. Los kernels también pueden tener un conjunto V de parámetros por valor. Hemos omitido este último conjunto ya que no está relacionado con las dependencias ni con con las tranferencias de datos asíncronas.
- **Lanzamiento de tareas de host:** $H(g, In, Out)$. Petición para ejecutar una función g en el host, con el mismo formato que una petición de lanzamiento de kernel, incluyendo los conjuntos de entradas y salidas.
- **Wait:** $W(x)$. Petición para bloquear la ejecución en el código principal del host hasta que todas las peticiones que involucren x hayan finalizado. En varios modelos de programación, como CUDA, esta operación está implícita por defecto después de una petición $DTH(x)$, sin embargo puede ser evitada por el programador mediante el uso explícito de transferencias asíncronas. En nuestro modelo podemos realizar una espera explícita para asegurar la portabilidad, y tener una semántica más clara en términos de sincronización. En la práctica, esta operación se utiliza únicamente cuando el código principal del programa necesita utilizar los valores del dispositivo. Por ejemplo, esta operación puede ser necesaria en un bucle con una condición de convergencia que es calculada con una operación de reducción en el dispositivo. También

```
int main(int argc, char *argv[]){
    ...
    __ctrl_block__(1)
    {
       //Controller creation
       Ctrl ctrl = Ctrl_Create( CTRL_TYPE_CUDA, DEVICE_ID );

       // Create data structures
       HitTile_float GxC = Ctrl_Alloc( ctrl, float, hitShape(3, 3) );
       HitTile_float GxR = Ctrl_Alloc( ctrl, float, hitShape(3, 3) );
       HitTile_float imgOrig = Ctrl_Alloc( ctrl, float, hitShape( rows, columns) );
       HitTile_float imgGy = Ctrl_Alloc( ctrl, float, hitShape( rows, columns) );
       HitTile_float imgGx = Ctrl_AllocDev( ctrl, float, hitShape( rows, columns) );
       HitTile_float imgTmp1 = Ctrl_AllocDev( ctrl, float, hitShape( rows, columns) );
       HitTile_float imgTmp2 = Ctrl_AllocDev( ctrl, float, hitShape( rows, columns) );

       // Domain of logical threads for the device
       Ctrl_Threads threads = Ctrl_Threads(rows, columns);

       // Initialization in the host and move to device
       Ctrl_HostTask( init, GxC, GxR );
       Ctrl_MoveTo( ctrl, GxC );
       Ctrl_MoveTo( ctrl, GxR );
       Ctrl_HostTask( getFrame, imgOrig );
       Ctrl_MoveTo(ctrl, imgOrig);

       // Sobel filter: Main loop
       // Clock: Synchronize and start measuring
       for (int index = 0; index < ITERATIONS; index ++) {
          Ctrl_Launch(ctrl, convolutionRow, threads, imgTmp, imgTmp2, imgOrig, GxR, GxC);
          Ctrl_HostTask( getFrame, imgOrig );
          Ctrl_MoveTo(ctrl, imgOrig);
          Ctrl_Launch(ctrl, convolutionColum, threads, imgGx, imgTmpGxC);
          Ctrl_Launch(ctrl, convolutionColum, threads, imgGy, imgTmpGxR);
          Ctrl_Launch(ctrl, saxpySqrt, threads, imgGx, imgGy, 1.0);
          Ctrl_MoveFrom(ctrl, imgGy);
          Ctrl_HostTask( putFrame, imgGy );
       }
       // Clock: Synchronize and stop measuring

       // Free the device and host memory
       Ctrl_Free( ctrl, GxC, GxR, imgOrig, imgGx, imgGy, imgTmp2, imgTmp2 );
       Ctrl_Destroy( ctrl );

    } //__ctrl_block__
    ...
} //main
```

Fig. 4. Fragmento del código principal del filtro de Sobel aplicado sobre un flujo de vídeo, programado utilizando la nueva implementación asíncrona de Controller.

se ha considerado una *operación global de espera* $global\_wait\_request\_W()$, sin parámetros, que bloquea la ejecución hasta que todas las peticiones previas han terminado.

B. Reglas para la ejecución asíncrona

En esta sección describimos la reglas que se aplican en tiempo de ejecución para decidir que operaciones deben esperar a estar listas para ser ejecutadas, y cuales pueden ser empezadas, realizándose en asíncrono, y pudiendo solaparse con otras operaciones previas o posteriores.

En el modelo de ejecución sínrono, todas las peticiones son ejecutadas en orden. Todas ellas se consideran operaciones bloqueantes, que paran la ejecución principal del programa hasta que terminan. La siguiente petición puede ser evaluada y preparada para evitar tiempos de inicialización, pero su ejecución no puede empezar hasta el final de la anterior.

En el modelo de ejecución asíncrono, esta regla es más flexible para permitir el solapamiento de algunas peticiones. La ejecución de kernels concurrentes es un problema ortogonal al solapamiento de transferencias y computación de kernels, por ello esta fuera del ámbito de este trabajo. Nuestra propuesta, preserva el orden secuencial de ejecución de los kernels. Estos son ejecutados uno tras otro en el mismo orden en el que se han realizado sus peticiones. De manera similar, las peticiones de ejecución de tareas de host también se realizan en orden, aunque pueden solaparse con los kernels si las dependencias lo permite. Las transferencias de datos pueden solaparse con las ejecuciones de ambas tareas de host y kernels.

La reglas que deciden cuando uan petición puede empezar se han diseñado mediante el análisis de dependencias entre los diferentes tipos de operaciones y los roles de entrada/salida de sus parámetros. Las operaciones de control de memoria (allocate/deallocate) y de espera (wait) siguen unas reglas más simples debido a su semántica. Los kernels, las tareas de host, y las transferencias de datos deben ser analizadas considerando como problemas con múltiples-

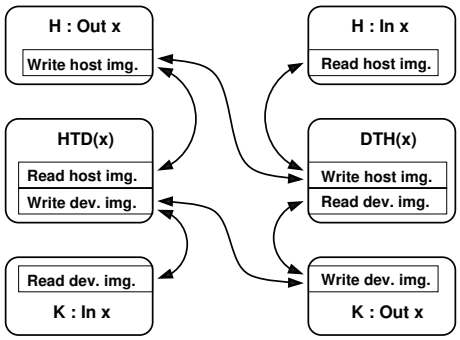

Fig. 5. Dependencias entre tipos de peticiones. Las cajas redondeadas identifican los tipos de peticiones que hay, usando x como parámetro. Se distinguen entre peticiones K o H que utilizan x con un rol de entrada o salida. Para simplificar el diagrama, no se muestran las peticiones de espera ni de gestión de memoria. Las peticiones que no están unidas por flechas se pueden ejecutar de forma concurrente.

lectores/múltiples-escritores. Cada estructura de datos puede tener dos imágenes de memoria, una en el host y otra en el dispositivo. Una operación de $HTD(x)$ lee la imagen de x en el host, y escribe en la imagen de x en el dispositivo. Una operación de $DTH(x)$ lee la imagen de x en el dispositvo, y escribe en la imagen de x en el host. Las operaciones K leen las imágenes en el dispositivo de los parámetros de entrada $x \in In$, y escriben en las imágenes en el dispositivo de los parámetros de salida $y \in Out$. Las operaciones H leen las imágenes en el host de los parámetros de entrada $x \in In$, y escriben en las imágenes en el host de los parámetros de salida $y \in Out$. Pueden haber múltiples lecturas concurrentes de una misma imágen de una estructura de datos. Varias peticiones de diferentes tipos pueden empezar si no existe una intersección entre sus parámetros, o si únicamente realizan lecturas sobre la misma imágen de los parámetros en común. Una petición r que escribe en una imagen de memoria debe esperar hasta que todas las peticiones anteriores de escritura o lectura hayan terminado. Igualmente, todas las peticiones posteriores que escriban o lean en esa imágen de memoria deben esperar por r. Este esquema de dependencias está reflejado en la figura 5.

Las reglas del modelo de ejecución asíncrona se pueden resumir de la siguiente forma:

- $Alloc(x)$ o $AllocDev(x)$: Todas las peticiones posteriores que involucren a x deben esperar hasta el final del alojamiento.
- $Free(x)$: Una petición para desalojar x debe esperar por todas las peticiones anteriores que involucren x.
- $HTD(x)$: Debe esperar si x (a) es un parámetro de una operación DTH; (b) aparece como parámetro (de entrada o salida) en un kernel anterior que no ha finalizado (operación K); (c) aparece como parámetro de salida de una tarea de host que no ha finalizado (operación H).
- $DTH(x)$: Debe espera si x (a) es un parámetro de una operación HTD; (b) aparece como parámetro de salida de un kernel que no ha finalizado (operación K); (c) aparece como parámetro (de entrada o salida) en una tarea de hots anterior que no ha finalizado (operación H).
- $K(f, I, O) : x \in I, y \in O$: Debe esperar si (a) x es un parámetro de una operación HTD sin completar; (b) y es un parámetro de una operación HTD o DTH sin completar; (c) hay una operación K previa sin completar.
- $H(g, I, O) : x \in I, y \in O$: Debe esperar si (a) x es un parámetro de una operación DTH sin completar; (b) y es un parámetro de una operación HTD o DTH sin completar; (c) hay una operación H previa sin completar.
- $W(x)$: Debe esperar por todas las operaciones previas que involucren a x.

Una petición está lista para empezar tan pronto como las condiciones de espera se hayan cumplido. Las peticiones pendientes deben ser encoladas hasta que las condiciones de disposición se cumplan. Conceptualmente, se pueden utilizar varias colas de ejecución para "ready-to-execute kernels", "ready-to-execute host codes", "ready-to-execute DTH transferences", y "ready-to-execute HTD transferences". Las operaciones de diferentes colas que estén preparadas pueden ejecutarse de forma asíncrona y pueden ser solapadas de forma segura.

C. La nueva interfaz asíncrona de Controller

Las nuevas funcionalidades definidas en el modelo asíncrono de Controller que realizan las transferencias son `Ctrl_MoveTo` para operaciones HTD, y `Ctrl_MoveFrom` para operaciones DTH. `Ctrl_Launch` se usa para invocar la computación de un kernel (operación K) como en el modelo síncrono de Controller. `Ctrl_HostTask` ha sido añadida para invocar las tareas de host (operación H). `Ctrl_Wait` implementa la operación de espera. `Ctrl_Alloc`, `Ctrl_AllocDev` y `Ctrl_Free` implementan la funciones de gestión de memoria para conjuntos de HitTiles. Las tareas de host que se quiera lanzar y ejecutar de forma asíncrona se deben declarar de forma similar a los kernels utilizando la primitiva `CTRL_HOST`.

El mismo programa de Sobel que se ha utilizado como ejemplo en la figura 2 se muestra en la figura 4 reescrita haciendo uso de la nueva interfaz del modelo asíncrono de Controller. El código de Controller se encuentra ahora contenido dentro de un bloque estructurado precedido por `__ctrl_block__()`. El parámetro indica el número de instancias de Controller que se crean y asocian a diferentes dispositivos dentro del bloque. Esta función sirve para engendrar y preparar hilos necesarios para el funcionamiento interno de cada instancia de Controller. Entre las líneas 9 y 15 se declaran y alojan las estructuras de datos con las nuevas operaciones, que también preparan la memoria del host para permitir transferencias asíncronas de memoria más eficientes (declarando la memoria como *pinned*). En el nuevo código, las operaciones de gestión de memoria únicamente aparecen al principio y al final del programa. Las transferencias del host al dispositivo y viceversa se entremez-

clan con invocaciones a tareas de host y kernels de dispositivo siguiendo el algoritmo de forma natural. Las transferencias se piden a la instancia de Controller tan pronto como las tareas de host y los kernels que generan los datos se encolan, para maximizar las oportunidades de solapamiento entre la computación y dichas transferencias.

D. Nueva política de gestión de la cola

La anterior versión del modelo de Controller define únicamente dos operaciones que involucraban trasnferencias de datos y gestión de memoria. Existen dos posibles políticas de manejo de memoria asociadas a ellas. La política *eager* mueve los datos desde/hacia el dispositivo cuando la operación se solicita. La política *lazy* mueve los datos al dispositivo exclusivamente cuando un kernel la usa por primera vez.

El modelo asíncrono de Controller introduce un rediseño de la gestión interna de la cola de peticiones. Las funciones de Controller que generan peticiones cuando son llamadas desde el código principal no son bloqueantes en el nuevo modelo, exceptuando las funciones de espera y de gestión de memoria gracias a su semántica. Todas las operaciones no bloqueantes de peticiones son encoladas en el Controller con la información sobre sus parámetros, y las referencias de las estructuras de HitTile.

La estructura de HitTile se ha expandido para almacenar información sobre su uso en operaciones de lectura y escritura para cada uno de los flujos conceptuales de las diferentes operaciones (transferencias de datos, K o H).

Hemos previsto dos módulos de políticas: Ejecución Síncrona y Asíncrona. El módulo de política síncrona simplemente bloquea la ejecución cada vez que extrae una petición y comienza su ejecución. El módulo de política asíncrona saca una petición de la cola pero se delega al backend la aplicación de las reglas presentadas en IV-B. Por lo tanto, se pueden aprovechar diferentes mecanismos de asincronía en distintos backends, utilizando las primitivas y recursos específicos por el modelo de programación correspondiente. Los métodos del backend utiliza la información dada en los parámetros y en los tiles aosciado para evaluar las reglas y transformar las peticiones en las llamadas a las funciones correspondientes del modelo de programación específico para el dispositivo.

Planteamos un sistema que traducelas dependencias generadas al aplicar las reglas descritas en IV-B a eventos o condiciones de espera, utilizando la interfaz de CUDA, para la implentación del backend del caso actual. Esto permite que el driver ejecute las peticiones en el orden adecuado de manera eficiente sin la supervisión del host.

La extensión de la estructura del HitTile contiene 6 campos, de tecnologías clásicas de concurrencia en el host y específicas de CUDA, para detectar el fin de la última operación que se realiza sobre el tile para diferentes propósitos:

- **DT_HTD:** Transferencia de datos, escritura en el dispositivo.
- **DT_DTH:** Transferencia de datos, lectura desde el dispositivo.
- **K_IN:** Parámetro de entrada en kernel, escritura en el dispositivo.
- **K_OUT:** Parámetro de entrada en kernel, lectura en el dispositivo.
- **H_IN:** Parámetro de entrada en tarea de host, escritura en el host.
- **H_OUT:** Parámetro de entrada en tarea de host, lectura en el host.

Este sistema implementa el modelo propuesto de ejecución asíncrona. Además, asegura que se respeta las ejecuciones de las peticiones en el orden correcto, siguiendo sus dependencias, y permitiendo el solapamiento de las operaciones en los diferentes streams, mediante el uso de la tecnología nativa de CUDA.

TABLA I
RESULTADOS DE LAS MÉTRICAS DE ESFUERZO DE DESARROLLO DE LOS CÓDIGOS DE REFERENCIA Y DE CONTROLLER. SE INCLUYE UNA COMPARATIVA ENTRE EL NÚMERO DE LÍNEAS DE CÓDIGO (LOC), NÚMERO DE TOKENS (TOK), COMPLEJIDAD CICLOMÁTICA DE MCCABE, Y MÉTRICA DE HALSTEAD (HALST.).

Case study	Version	LOC	TOK	CCN	Halst.
Sobel filter	Ctrl	124	1 337	10	625 156
	CUDA_Syn	113	1 230	15	1 025 481
	CUDA_Asyn	231	1 863	24	1 265 867
	OpenCL_Syn	266	2 423	22	33 645
	OpenCL_Asyn	312	2 554	22	34 879

V. ESTUDIO EXPERIMENTAL

En esta sección, describimos el estudio experimental realizado para la evaluación de las ventajas y limitaciones potenciales del modelo de ejecución asíncrona en la librería de Controller. Esta sección incluye: (1) Una descripción del caso de estudio considerado; (2) un estudio de rendimiento de nuestra propuesta; y (3) una comparativa del esfuerzo de desarrollo entre la programación usando la interfaz asíncrona de Controllers o usando CUDA, u OpenCL.

A. Casos de estudio

Hemos seleccionado el programa del filtro de Sobel para probar nuestra aproximación e implementación. Este programa trabaja con valores reales de precisión simple (o *floats*). El programa aplica el filtro de Sobel a un flujo de vídeo en el dispositivo (ver figura 4). Por cada iteración se envía un fotograma al dispositivo y se retorna el resultado. Este ejemplo permite probar el solapamiento de transferencias en ambos sentidos. Para simplificar, no se tienen en cuenta las operaciones de host para las mediciones en el ejemplo para enfocar el caso en el solapamiento de las transferencias y los kernels. El programa se ha probado con imágenes Ultra-HD de tamaños de 4K, 8K y 16K.

B. Entorno de experimentación

En esta experimentación comparamos la implementación de Controller, con las implementaciones síncrona y asíncrona de programas de referencia. El código de Controllers es el mismo para ambos modos de ejecución. Ambas políticas de ejecución se puede seleccionar en tiempo de ejecución.

Los experimentos se han ejecutado sobre una plataforma equipada con una GPU de Nvidia. La máquina anfitriona se compone de un procesador Intel i5-3330 @3.0GHz, con 8 GB GDDR3 de memoria principal, y esta equipada con una NVIDIA's Tesla K40c, con 2880 cores @ 745 MHz, y 12 GB GDDR4 de memoria global. El S.O. es una distribución Linux CentOS 7. Todos los programas se han compilado con GCC v7.2, utilizando el toolkit y el driver de CUDA 10.0, y OpenCL 1.2. Por simplicidad, los tamaños de bloques usados en la ejecuciones de los kernels de GPU son de 16×16. La experimentación de rendimiento mide el tiempo de reloj desde el comienzo hasta el final del bucle principal del programa. Este tiempo incluye las transferencias de datos, la computación, y los sobrecostes del sistema.

C. Estudio de rendimiento

Los tiempos de ejecución del programa del filtro de Sobel se pueden observar en la tabla II. El programa del filtro de Sobel no mide las operaciones de host, y se ha enfocado en las transferencias de datos en ambas direcciones. La mejoría observada proviene del solapamiento de los kernels, las transferencias de datos en ambas direcciones. En la figura 6 se muestra gráficamente la representación de la ejecución que se obtiene del profiler visual de CUDA de la ejecución de ambas versiones de Controllers. El solapamiento de las transferencias de datos en ambas direcciones provoca que tarden un poco más de tiempo, pero permite un mayor solapamiento de ambas transferencias y los kernels. Este hecho lleva a altas mejoras en el rendimiento del modo asíncrono.

Comparando los resultados de Controller con las correspondientes referencias (síncrona o asíncrona), las mediciones indican un pequeña y estable penalización de tiempo, debido a la gestión del sistema de cola y los mecanismos de sincronización de Controller.

D. Métricas de esfuerzo de desarrollo

Esta sección analiza las diferencias en el esfuerzo de desarrollo entre los códigos de Controller y las implementaciones base utilizando CUDA para los escenario asíncronos. Hemos medido las siguientes métricas de esfuerzo de desarrollo: Número de líneas de código; Número de tokens; Complejidad Ciclomática de McCabe [16] y la métrica de desarrollo de Halstead [17]. Las primeras métricas midel el volumen de código que el programador debe desarrollar. La tercera cuantifica de forma racional el esfuerzo necesario para programar en términos de divergencias de código ye incidencias potenciales que pueden ser consderadas en el desarrollo, prueba y debug del programa. La última métrica se basa en la complejidad y volumen de código para obtener una medición del esfuerzo de desarrollo. Los códigos empleados incluyen las definiciones de los kernels, las caracterizaciones de los kernels, el código principal del host, y las estructuras de datos empleadas. Hay un importante volumen de código dentro de los kernels, y es igual para todas las versiones. Los códigos de tareas de host del filtro de Sobel de han excluido del estudio.

Los resultados mostrados en la tabla I indican que la programación usando Controllers genera un menor volumen de código, y reduce las métricas de Halstead. Esto es se puede observar especialmente en que las versiones base asíncronas introducen manualmente mencanismos más complejos de sincronización. Estos mecanismos son transparentes en los programas de Controllers. Un análisis más en detalle revela una alta reducción de las métricas en las partes de código del host correspondientes al flujo principal del programa, como se esperaba. Estos resultados indican un menor esfuerzo de desarrollo necesario en lo programas de Controllers en comparación de usar CUDA u OpenCL directamente.

VI. Conclusiones

Este trabajo persenta una técnica para detectar y ejecutar de forma transparente y asíncrona transferencias de datos entre una máquina anfitriona y un acelerador de harware. La implementación extiende las funcionalidades de Controller, rediseñando la estructura interna, e incluyendo una nueva gestión del sistema de colas con políticas que soportan modos de ejecución síncrona y asíncrona. Este modo se puede elegir en tiempo de ejecución. Se han desarrollado dos backends, uno utilizando CUDA, y otro con OpenCL, que demuestran como esta técnica puede ser aplicada sobre un amplio rango de dispositivos.

La implementación propuesta solapa automáticamente las transferencias de memoria con la computación de kernesl y la ejecución de código de host, sin añadir una complejidad significativa. El estudio experimental muestra mejoras notables en el rendimiento de programas de filtro sobre los fotogramas de un vídeo con transferencias intensivas en los ambos sentidos. Las métricas de desarrollos indican que los códigos de Controllers son más sencillos de desarrollar que una implementación manual con las librerías y herramientas propias de los dipositivos utilizados.

Como trabajo futuro se incluye el desarrollo de nuevos backends para soportar otros tipos de dispositivos con diferentes restricciones y otras políticas de gestión de colas más sofisticadas.

Referencias

[1] TOP500.org, "Top500 Supercomput. Sites," Dec 2017, on `http://www.top500.org`.

[2] Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos, "Mpi-aware compiler optimizations for improving communication-computation overlap," in *23rd Int. Conf. on Supercomput.*, New York, NY, USA, 2009, ICS '09, pp. 316–325, ACM.

[3] Ana Moreton–Fernandez, Hector Ortega–Arranz, and Arturo Gonzalez–Escribano, "Controllers: An abstraction

TABLA II

Tiempos de ejecución del programa del filtro de Sobel. La última columna muestra el porcentaje de mejora de rendimiento entre el modos asíncrono sobre el modo síncrono de Controller.

Sobel filter (CUDA-Backend on NVIDIA GPU)

Input Size	CUDA-Syn	CUDA-Asyn	Ctrl-Syn	Ctrl-Asyn	% Improvement
4 096 × 2 160	0,975	0,586	0,984	0,592	39,8
7680 × 4 320	3,734	2,140	3,760	2,154	42,7
15 360 × 8 640	14,950	8,396	15,047	8,452	43,8

Sobel filter (OpenCL-Backend on NVIDIA GPU)

Input Size	OpenCL-Syn	OpenCL-Asyn	Ctrl-Syn	Ctrl-Asyn	% Improvement
4 096 × 2 160	1,222	0,675	1,228	0,680	44,6
7680 × 4 320	4,596	2,527	4,584	2,541	44,5
15 360 × 8 640	18,382	10,202	18,276	10,152	44,4

Sobel filter (OpenCL-Backend on AMD GPU)

Input Size	OpenCL-Syn	OpenCL-Asyn	Ctrl-Syn	Ctrl-Asyn	% Improvement
4 096 × 2 160	0,597	0,595	1,122	0,705	37,1
7680 × 4 320	2,166	2,127	3,377	2,270	32,6
15 360 × 8 640	8,718	8,406	14,786	8,702	40,8

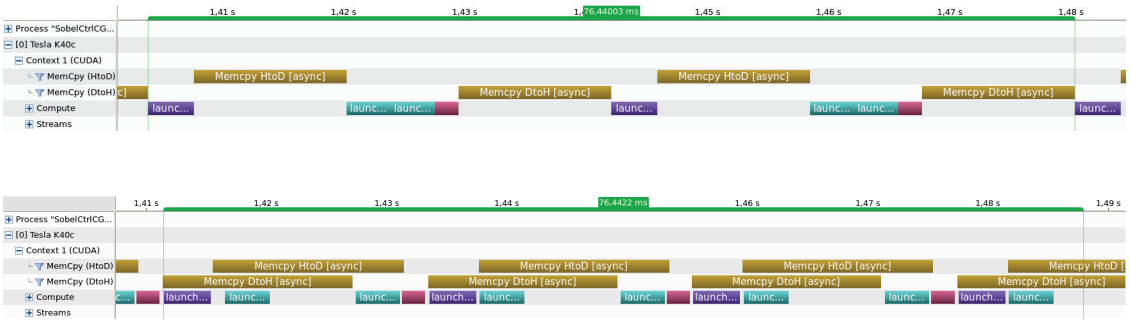

Fig. 6. Capturas de pantalla de la herramienta CUDA 10.0 Visual Profiler mostrando 76ms. de ejecución del programa de Controller del filtro de Sobel. Arriba: modo de ejecución síncrono. Abajo: modo de ejecución asíncrono con solapamientos entre transferencias, kernels y tareas de host.

to ease the use of hardware accelerators," *The International Journal of High Performance Computing Applications*, vol. 0, no. 0, pp. 16, 2017.

[4] Ana Moreton-Fernandez, Eduardo Rodriguez-Gutiez, Arturo Gonzalez-Escribano, and Diego R. Llanos, "Supporting the Xeon Phi coprocessor in a heterogeneous programming model," in *Euro-Par 2017: 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 – September 1, 2017, Proceedings*, Santiago de Compostela, Galicia, Spain, 2017, pp. 457–469, Springer International Publishing.

[5] 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 *18th Int. Conf. on Computational Science and Engineering*, Porto, Portugal, Oct 2015, pp. 17–26, IEEE.

[6] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers, "Civl: The concurrency intermediate verification language," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2015, SC '15, pp. 61:1–61:12, ACM.

[7] Antonio Vilches, Angeles Navarro, Francisco Corbera, Andres Rodriguez, and Rafael Asenjo, "heterogeneous parallel for template based on tbbs," in *Proceedings of the 10th International Symposium on High-Level Parallel Programming and Applications*, Valladolid, Spain, 2017, Springer International Publishing.

[8] 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.

[9] T. Gysi, J. Bär, and T. Hoefler, "dcuda: Hardware supported overlap of computation and communication," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, Utah, EE. UU., Nov 2016, pp. 609–620, IEEE.

[10] R. Vasudevan, Sathish S. Vadhiyar, and Laxmikant V. Kalé, "G-charm: An adaptive runtime system for message-driven parallel applications on hybrid systems," in *ICS 2013 - Proceedings of the 2013 ACM International Conference on Supercomputing*, Eugene, Oregon, United States, 7 2013, pp. 349–358, ACM.

[11] C++ Executors, "C++ Standards Committee Papers," 2018, `goo.gl/bf4evL`,Last visit: August, 2018.

[12] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt, "Gdev: First-class gpu resource management in the operating system," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, Berkeley, CA, USA, 2012, USENIX ATC'12, pp. 37–37, USENIX Association.

[13] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke, "VAST: the illusion of a large memory space for GPUs," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, New York, NY, USA, 2014, PACT '14, pp. 443–454, ACM.

[14] Rafael C. Gonzalez and Richard E. Woods, *Digital Image Processing (3rd Edition)*, Prentice Hall, New Jersey 07458, 3 edition, Aug. 2007.

[15] 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, 2014.

[16] Thomas J McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. 4, pp. 308–320, 1976.

[17] Maurice H Halstead, *Elements of Software Science (Operating and programming systems series)*, Elsevier Science Inc., New York, NY, USA, 1977.