Técnicas de implementación de Stencils en multi-GPU distribuidas

Senmao Ji Ye [1] , Arturo González Escribano[2] , Diego R. Llanos[3]

Resumen— **En el patrón de computación denominado *stencil* cada elemento de una estructura de datos de tipo array se actualiza iterativamente en función de los valores de sus vecinos. Entre otras aplicaciones, este patrón permite resolver numéricamente sistemas de ecuaciones en derivadas parciales, por lo que es de gran interés en el computo científico, creciendo incesantemente los requerimientos de tamaño de datos y carga computacional en problemas reales. La estructura de este patrón permite utilizar estrategias sencillas de paralelismo de datos, por lo que su paralelización, tanto en CPUs como en aceleradores es de gran interés. Sin embargo, la necesidad de sincronización y comunicación entre elementos de proceso deriva en problemas relacionados con la capacidad de distribuir la carga y explotar multiples dispositivos simultaneamente. En este trabajo presentamos un repaso y actualización de técnicas de programación eficientes basadas en MPI y CUDA para explotar este patrón de computación en sistemas multi-GPU distribuidos. Nuestros resultados muestran cómo las técnicas utilizadas pueden aliviar los problemas de comunicación entre host y GPUs, obteniendo rendimientos y escalabilidad en función de las capacidades del sistema de interconexión entre nodos.**

Palabras clave— **Stencil, Multi-GPU, computación distribuida**

I. Introducción

PARA la resolución de numerosos problemas científicos actuales, en los que se aplican ecuaciones derivadas parciales, tales como el cálculo de la propagación del sonido o del calor, se utiliza una técnica de computación llamada Stencil. Una técnica Stecil consiste en una iteración temporal en la que en cada iteración se actualizan todas y cada una de las celdas de una matriz multidimensional, a partir de los valores anteriores de un conjunto de celdas vecinas. Cada problema define cuántas celdas vecinas se utilizan y qué pesos se aplican a cada una antes de aplicar la función que calcula el nuevo valor de la celda que está siendo procesada.

La utilidad de este patrón y su estructura que permite la aplicación de técnicas sencillas de paralelismo de datos, hace que haya sido ampliamente estudiado en la literatura. Desde implementaciones sencillas ilustrativas de técnicas básicas de programación distribuida (p.e. [1]), o versiones optimizadas en aceleradores hardware (p.e. [2]). Se han propuesto técnicas para su explotación eficiente en sistemas multi-GPU (p.e. [3]), o en clústers heterogéneos (p.e. [4]).

En este trabajo presentamos un repaso actualizado de técnicas de implementación de stencils para sistemas distribuidos de aceleradores GPU, orientadas ocultar las latencias de las comunicaciones necesarias en cada iteración temporal. Utilizando como caso de estudio un stencil básico con una carga de cómputo mínima por elemento (el método de Jacobi), mostramos con diferentes aproximaciones técnicas eficientes y poco complejas para solapar computación y comunicación. Estudiamos el impacto de las copias asíncronas entre GPU y host, y entre nodos de la red, exploramos funcionalidades del API de CUDA no utilizadas en estudios previos, y exploramos la escalabilidad en presencia de tecnologías de red asequibles y de fácil acceso. Nuestros resultados muestran cómo las técnicas utilizadas pueden aliviar los problemas de comunicación entre host y dispositivos aceleradores, así como entre nodos, para problemas de tamaños adecuados para los espacios de memoria de los dispositivos aceleradores actuales.

El resto del artículo se organiza de la siguiente forma. En la sección 2 se muestra una descripción más detallada de los patrones stencil y del caso de estudio. En la sección 3 se describen las técnicas utilizadas. En la sección 4 se presentan un estudio experimental para determinar la utilidad de las técnicas dicutidas. La sección 5 presenta las conclusiones y describe el trabajo futuro.

II. Computación Stencil

Los programas basados en computación de tipo *stencil* repiten iterativamente la actualización de todas las celdas de un array multidimensional en función de los valores anteriores de celdas vecinas hasta una condición de terminación que puede depender de los valores calculados. Las *celdas vecinas* se definen en función de su posición relativa respecto al valor de los índices de cada dimensioń de la celda que se computa.

Cada problema define un patrón o *stencil*, que indica qué celdas se consideran vecinas, aportando su valor a la función que calcula el nuevo valor de una celda, y opcionalmente pesos a aplicar dependiendo de la posición relativa de la cada celda vecina respecto a la que se está computando. Uno de los métodos stencil más simples y estudiados es el método de Jacobi (ver Fig. 1). En este código se consideran como celdas vecinas las que tienen índices multidimensionales que varían con respecto a los de la celda actual en sólo una unidad, en un único índice. La función de actualización es la media aritmética de los valores vecinos, todos con el mismo peso.

El patrón o stencil se puede complicar con celdas vecinas a mayor distancia (radio del stencil), de

[1]Dpto. de Informática, Univ. Valladolid, España. e-mail: Senmao.ji@icloud.com.
[2]Dpto. de Informática, Univ. Valladolid, España. e-mail: Arturo@infor.uva.es.
[3]Dpto. de Informática, Univ. Valladolid, España. e-mail: Diego@infor.uva.es.

forma no equilibrada en las diferentes dimensiones o ejes [5], hasta llegar a métodos complejos con patrones que se alteran de forma dinámica mientras avanzan las iteraciones.

A. Jacobi 2D

```
for (i=1; i<rows-1; i++) {
 for (j=1; j<columns-1; j++) {
  matrix[i][j] =
   ( matrixCopy[i-1][j] +
     matrixCopy[i+1][j] +
     matrixCopy[i][j-1] +
     matrixCopy[i][j+1] ) / 4;
 }
}
```

Fig. 1. Codigo de Jacobi en 2D

El método Jacobi aplicado a una matriz de dos dimensiones (ver Fig. 1) calcula el nuevo valor de cada celda cómo la media aritmética de los valores anteriores de los cuatro vecinos adyacentes. Utilizando una estructura de apoyo para almacenar los valores de todos los elementos de la matriz en la iteración anterior, este proceso es ideal para ser realizado en paralelo, ya que todos los valores de salida son independientes entre sí.

La dependencia del cálculo de una celda, con los valores de las vecinas calculados en la iteración anterior, implica que en ejecuciones con multiples procesos, un proceso requiera para el cómputo de la parte que se le asigna, de valores que se encuentran en un proceso distinto. Para poder calcular los valores de las celdas situadas en los bordes de las submatrices de cada proceso (boundary), una solución típica es ampliar las submatrices locales con una fila/columna de celdas en cada dirección, para almacenar los datos adyacente a las boundaries, que han sido calculados en el proceso vecino en la iteración anterior (ver Fig. 3). Estas celdas adicionales reciben el nombre de *halo*. Tras calcular todo los valores de la submatriz, cada proceso comunica los datos actualizados de sus bordes (boundaries), enviándolos hacia los halos vecinos. que se encuentran en otros procesos (E), recibiendo a su vez los datos de los bordes (boundaries) de los procesos vecinos, vecinas (A)(B)(C) y (D), que se almacenarán en los halos locales.

En ejecuciones con un único dispositivo, esta dependencia no nos produce muchos inconvenientes, ya que disponemos de toda la matriz en la misma memoria global, por lo que se puede acceder desde cualquier punto independientemente de por quién ha sido actualizado en la iteración anterior. El único requisito es una barrera de sincronización de los elementos de proceso implicados entre una iteración y otra. Otras técnicas asociadas a la explotación de la jerarquía de memoria son posibles. Por ejemplo, en uno de los programas de ejemplo que acompañan al Toolkit de CUDA suministrado por NVIDIA, que implementa una convolución de imagen basada en el método de Jacobi, cada bloque de hilos realiza la copia local de la submatriz correspondiente en memoria compartida, ademas del halo de dicha submatriz [6]. Los accesos a memoria compartida son mucho más rápidos, con lo que se aceleran los accesos a los elementos cuando se reutilizan, en el momento en que se calcula cada elemento vecino.

```
__shared__ float sData
[THREADS_BLOCK_Y + 2 * RADIUS]
[THREADS_BLOCK_X + 2 * RADIUS];

unsigned int index = (i)* Nj + (j) ;

if( li<RADIUS ) // copy top and bottom halo
{
//Copy Top Halo Element

// Boundary check
if(blockIdx.y > 0)
sData[li][e_lj] =
    input[index - RADIUS * Nj];

//Copy Bottom Halo Element
// Boundary check
if(blockIdx.y < (gridDim.y-1))
sData[e_li+THREADS_BLOCK_Y][e_lj] =
    input[index + THREADS_BLOCK_Y * Nj];

}

if( lj<RADIUS ) // copy left and right halo
{
    // Boundary check
if( blockIdx.x > 0)
sData[e_li][lj] = input[index - RADIUS];

// Boundary check
if(blockIdx.x < (gridDim.x-1))
sData[e_li][e_lj+THREADS_BLOCK_X] =
    input[index + THREADS_BLOCK_X];
}
```

Fig. 2. Extracto del coígo de Jacobi 2D implementado en un ejemplo de convolución de imagen suministrado con el Toolkit de CUDA

Para la implementación en un clúster multi-GPU distribuido, sera necesario utilizar una API para el paso de mensajes entre procesos, como por ejemplo MPI (Message Passing Interface). También será necesario utilizar una API para la implementación del paralelismo en el dispositivo o dispositivos locales. En este trabajo, utilizaremos MPI como la API empleada para el paso de mensajes, y CUDA como la API para explotar los dispositivos GPU.

El método de comunicación entre procesos por cada iteración que evaluamos en este trabajo es el siguiente:

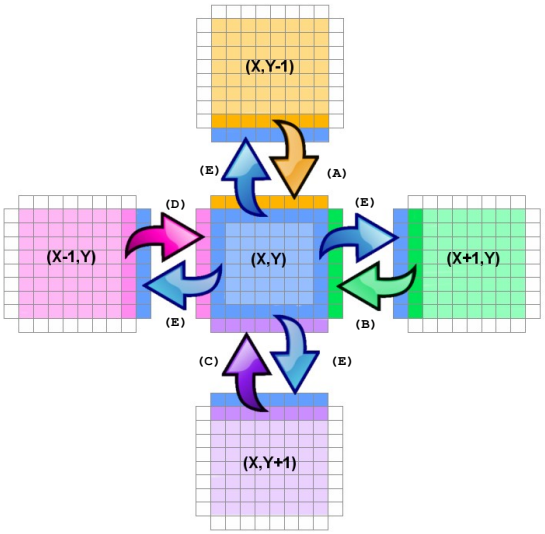

Fig. 3. Comunicación entre procesos en Jacobi 2D. Se muestran los halos o extensiones de la submatriz asignada al proceso, y los movientos de datos entre bordes y halos.

1. Se programan cinco kernels para computar la submatriz del proceso asignada a un dispositivo. Un kernel se encargara de la parte central, sin incluir los bordes. El resto de kernels computaran cada uno un borde (arriba, abajo, derecha, izquierda), en caso de que exista. Al repartir la matriz entre los dispositivos, aparecen submatrices que tocan alguno de los bordes. Dichas submatrices no tienen halo en esa dirección, ni necesidad de comunicar o procesar esos bordes de forma especial, ya que no hay otro proceso remoto que necesite esos datos. Cada uno de los cinco kernels se puede ejecutar de forma asíncrona. En CUDA se pueden utilizar *streams* o colas de instrucciones diferentes para lanzar cada kernel para conseguir dicha asincronía, ejecutando todos los kernels de forma concurrente.
2. En el stream (cola de instrucciones) donde se lanza la ejecución de cada kernel que computa un borde, encolamos inmediatamente después una operación de copia de los datos almacenados para ese borde, en la memoria del dispositivo, hacia la memoria del host. De esta forma, en el momento en que acabe la computación de un borde, comienza la copia asíncrona de los nuevos datos hacia el host. Utilizamos la función *cudaMemcpyAsync*.
3. Transferencias de datos de los bordes entre dispositivo y host. Estamos trabajando en C, con matrices almacenadas en lo que se conoce como *row major order*. Los elementos se almacenan por filas, de forma que los elementos de columnas consecutivas dentro de la misma fila, están consecutivos en memoria. Por tanto, el envio de los bordes horizontales (parte de una fila) se realizan directamente con una llamada a la función de copia asíncrona de CUDA, pasando el puntero de comienzo del borde y su tamaño.
   En cambio, para los bordes laterales (verticales), no podemos realizar el paso de memoria de esta manera ya que los elementos del borde no están consecutivos en la memoria. Mover dato a dato con diferentes operaciones de transferencia de memoria es muy ineficiente. En la siguiente sección se discuten dos mecanismos diferentes para mejorar estas transferencias.
   Finalmente se realiza una sincronización o espera para asegurar que los datos de los cuatro bordes se han transferido a la memoria del host. Se puede realizar con llamadas a la función *cudaStreamSynchronize* con cada uno de los cuatro streams asociados a las operaciones sobre los bordes.
4. Comunicación de datos de bordes entre procesos MPI. Las operaciones de comunicación en MPI pueden ser muy costosas, especialmente entre procesos asignados a nodos de red diferentes. Es fundamental que estas operaciones se realicen de forma asíncrona, pudiendo solaparse con la computación principal de la submatriz central en el dispositivo.
   Dado que el kernel que ejecuta la parte central en el dispositivo ya ha sido lanzado, cualquier tipo de operaciones de comunicación entre vecinos es viable. En este trabajo utilizamos funciones *MPI_Isend, MPI_Irecv* para el envío y recepción de los bordes hacia los halos remotos. Se inician las operaciones de envío y recepción y se realiza una sincronización con *MPI_Waitall* para esperar a que se reciban los bordes remotos en los halos locales. La utilización de operaciones asíncronas permite que los mensajes se procesen en cuanto están disponibles, independientemente del orden en que se llama a las funciones de envío o recepción.
5. Se procede a transferir los bordes que acaban de llegar a los halos a las correspondientes imagenes en la memoria del dispositivo. Se utilizan de nuevo las técnicas comentadas para la transferencia inversa.
6. Se realiza una sincronización global del dispositivo para asegurar que el kernel que ejecuta la computación de la parte principal en el dispositivo ha terminado. Utilizamos una llamada a la función *cudaDeviceSynchronize*.
7. El ejemplo escogido ejecuta un número determinado de iteraciones. Si se desea introducir una condición de terminación basada en el valor de residuos, debería ser implementada en este punto.
8. Finalmente, se procede a intercambiar los punteros de las estructuras de datos de las dos matrices en el dispositivo, para evitar tener que realizar una copia de la matriz con los nuevos datos a la matriz donde se mantienen las copias de la iteración anterior.
9. Tras la finalización del bucle de iteracions, se se comprueba el número de iteraciones realizadas, para conocer donde se encuentra la matriz resul-

tado tras los cambios de punteros, efectuando la correción de punteros si es necesario. La matriz resultado final puede ser transferida a la memoria del host para su escritura en fichero, o su utilización en posteriores fases de la aplicación.

Este método asíncrono nos permite realizar las opraciones de forma agregada, con la mayor granularidad posible, manteniendo la mayor concurrencia posible, y solapando computación y comunicación para reduciendo notablemente el tiempo comparado con una implemtación síncrono más directa y simple.

III. Implementación y optimizaciones

Para mejorar las transferencias de memoria y reducir el ruido en las mediciones de los tiempos, se han aplicado distintas técnicas para optimizar la implementación de la solución comentada en la sección anterior.

A. *Transferencia de bordes verticales*

Discutimos dos posibles técinas para evitar que la transferencia de los bordes verticales entre host y dispositivo, cuyos elementos no son consecutivos en memoria, utilizando diferentes llamadas a funciones de transferencia para cada elemento.

La primera está basada en utilizar buffers extra para copiar los datos de cada borde en una memoria contigua antes de las transferencias. Esto nos obliga a realizar manualmente la programación de estas operaciones de marshalling/unmarshalling en kernels y en el host. Reservar manualmente espacio adicional en la memoria de host y dispositivo para dos matrices unidimensionales (buffers), uno de envío y otro de recpción, por cada borde vertical. El kernel que computa un borde vertical, copia los resultados directamente en la correspondiente posicón del buffer contiguo correspondiente (marshalling). Es necesario programar un kernel para realizar la operación inversa (unmarshalling) con los datos de los halos verticales recibidos desde procesos remotos, colocando los datos en sus posiciones correspondientes en la matriz de datos.

La segunda se basa basada en utilizar las funciones de copia de matrices bidimensionales incluídas en el API de CUDA. En concreto, la función *cudaMemcpy2D* permite a través de sus parámetros indicar el salto que hay entre los elementos discontiguos, transfiriendo una submatriz con menos columnas de las que están reservadas en la matriz completa. En el estudio experimental comparamos ambas opciones en términos de rendimiento.

B. *Memoria* pinned

La trasferencia de datos del host al dispositivo y viceversa, implica un coste de tiempo para realizar la comunicación a través de los buses PCI en los que se alojan las tarjetas GPU. Puede variar según cómo se reserve la memoria en el host. Por defecto, la memoria que reservamos en el host con las rutinas *alloc* de C, es memoria paginada identificada por una Ummmm... estas seguro de todo esto? No es sólo que tiene que hacer los accesos resolviendo las direcciones de memoria virtual a física? . Esto provoca que para realizar una transmisión de datos desde el dispositivo al host sea necesaria la reserva de un bloque de memoria no paginada, seguido de una copia en el host desde la memoria física a la recién reservada, sumándole el coste de la transmisión, espera y liberacion de memoria de todo este proceso. En cambio, si reservamos memoria *pinned* (memoria no paginada), la comunicacion es mas rápida debido a que la GPU no necesita hacer la resolución de la direccion de memoria, ya que la memoria pinned utiliza una dirección de memoria física (RAM), y se puede realizar directamente la transmisión de datos sin necesidad del sobrecoste en tiempo mencionado anteriormente.

C. *Afinidad entre CPU y GPU*

Otro factor importante para optimizar las transferencias de memoria es definir la afinidad del proceso/thread que está ejecutando el código del host, de forma que se ejecute en los cores que están en el mismo nodo NUMA que está conectado al bus PCI donde está conectada Esto permite evitar el sobrecoste de la comunicación entre distintos nodos de la CPU. No controlar esta afinidad nos puede producir resultados estocásticamente peores en los tiempos de transferencia durante una serie de pruebas del mismo programa. El core de la CPU que se encarga de procesar el programa por defecto lo escoge el sistema operativo en función de parámetros no tienen en cuenta el uso de los dispositivos externos. Además, si no se define la afinidad, el sistema operativo también puede decidir migrar el proceso a otros cores. En el ejemplo de la figura 4 hay una buena afinidad entre la CPU0 y las GPU0 y GPU1, ya que se encuentran en un mismo nodo NUMA y la memoria gestionada por sus cores estará normalmente reservada en los bancos de memoria más próximos a ella. Por tanto, las comunicaciones entre esa memoria y el bus PCIe de esa CPU es más eficiente que si tiene que trasmitirse a través de la conexión entre la CPU0 y la CPU1 para dirigirse a las GPUs que se encuentran en el otro nodo NUMA.

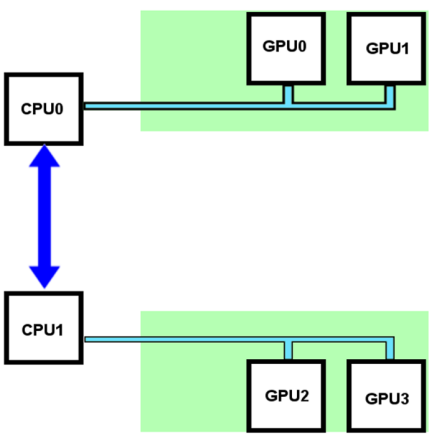

Fig. 4. Afinidad entre CPU y GPU

	Hydra	Chimera	Phoenix	Thunderbird
CPU	Xeon E5-2690v3	Xeon E5-2620v2	Core2 Quad Q6600	Core i5330
Cores	12	24	4	4
Clock	1.9 GHz	2.1 GHz	2.4 GHz	3 GHz
Memory	64 GB	32 GB	6 GB	8 GB
Num.GPUs	4	1	1	1
Model	GTX Titan Black	GTX Titan Black	Tesla K40c	Tesla K40c
Cores	2880	2880	2880	2880
Clock	980 MHz	980 MHz	745 MHz	745 MHz
Memory	6 GB	6GB	12 Gb	12 Gb

TABLA I
CLÚSTER TRASGO: MÁQUINAS Y CARACTERÍSTICAS

IV. EXPERIMENTACIÓN

En esta sección presentamos los resultados de un estudio experimental resalizado para comprobar el impacto realativo de las diferentes opciones y optimizaciones consideradas, y para verificar el nivel de escalabilidad conseguido por la solución. Este está en gran medida asociado a la capacidad de solapar la computación y comunicación de forma sostenida a lo largo de las iteraciones del programa.

Hemos escogido una implementación de Jacobi2D, con una partición de datos clásica y sencilla: bloques bidimensionales homogeneos. Este problema es uno de los stencils más sencillos en dos dimensiones, con una carga computacional muy baja por cada elemento considerado, con un esquema de comunicación entre vecinos con hasta cuatro vecinos por proceso. Esta disposición, con baja carga computacional por tarea, hace muy visible el impacto de las comunicaciones. Además, disponemos de implementaciones sencillas y eficientes de referencia. Por si al final metiste el que has puesto en la figura. Borra la frase que no sea cierta. // Hemos escogido como kernel de base la implementación contenida en el ejemplo distribuido con el Toolkit de CUDA en su versión 8.0. // Hemos escogido como kernel de base una traducción directa de la función de actualización de un elemento como se muestra en la figura **??**. (Y pones el código del kernel básico, que son pocas líneas).

Utilizamos como versión de referencia una implementación de la parte de comunicación en MPI basada en el ejemplo presentado en [1]. En nuestra versión de referencia la parte computacional se ejecuta en la GPU, lanzado un único kernel para calcular toda la submatriz asignada al proceso en cada iteración. Se realizan las transferencias de memoria para mover los datos de los bordes al host, o los halos recibidos al dispositivo, de forma síncrona con la función *cudaMemcpy*. Comparamos los resultados de rendimiento con la versión modificada según las especificaciones comentadas en la sección II, probando las difernentes opciones y optimizaciones detalladas en la sección III.

En este trabajo nos centramos en la escalabilidad de este problema en clústers de GPUs distribuidas. utilizan prácticamente toda la memoria global de los dispositivos disponibles, maximizando el tamaño global del grid computado. El tipo base de los arrays es *float*, para poder aprovechar el máximo número de unidades funcionales en las GPUs. En la experimentación utilizamos tamaños de problema que derivan en una ocupación de la memoria global de los dispositivos que va desde aproximadamente de un 5% a un 75% para las GPUs de referencia Titan Black.

A. Plataformas

Los experimentos se han realizado en dos clústers multi-GPU diferentes, ambos gestionados con un sistema de colas Slurm. El primero está gestionado directamente por el grupo de investigación *Trasgo* de la Universidad de Valladolid. Se trata de un clúster heterogeneos con nodos de diferentes capacidades. El la tabla I se resumen las máquinas que componen el clúster y sus características, incluyendo las de las GPUs instaladas. La tecnología de red de interconexión es Ethernet 1Gb entre los dos primeros nodos (Hydra y Chimera), y de 1Mb con los otros dos (Phoenix y Thunderbird).

La segunda plataforma es el clúster multi-GPUs del *Centro Extremeño de Tecnologías Avanzadas (CETA-Ciemat)*. Cada nodo tiene dos CPUs Intel Xeon E5-2620, con seis cores a 2.0 GHz, with 32 GB de RAM, y dos GPUs ..... modelo y caracteristicas..... La tecnología de red de interconexión es Infiniband QDR (40Gb/s) y FDR (56Gb/s).

B. Comunicación síncrona y asíncrona

Una de las optimizaciones com más impacto en computos iterativos que necesitan trasferir datos entre host y dispositivos en cada iteración, es el solapamiento del computo con la comunicación. Esto es factible en el caso de nuestra computación stencil, que no presenta dependencias entre iteraciones si se mantiene una copia con los datos de la iteración anterior. Por tanto es posible realizar las transferencias de datos y comunicaciones entre nodos de forma asíncrona.

En la tabla II presentamos el desglose de tiempos de ejecución dedicados a computación y a transferencias de memoria y/o comunicación entre nodos para diferentes tamaños de problema en la versión de referencia. Los tiempos están medidos en En que máquina o combinación de máquinas? Esto es ejecutando en cores o en GPUs??. Observamos que los tiempos de computo y comunicación son muy similares, creciendo ambos proporcionalmente con el

TABLA II
TABLA CON LOS TIEMPOS DE COMPUTACIÓN Y COMUNICACIÓN, CON 50 ITERACIONES (EN SEGUNDOS)

Tamaño matriz	Cómputo	Comumicación
6000x6000	0.3104	0.3263
20000x20000	3.0809	3.1852
25000x25000	5.0184	5.2029

TABLA III
TABLA CON MODELO SÍNCRONO Y ASÍNCRONO, CON 50 ITERACIONES En que máquina o combinación de máquinas?? (EN SEGUNDOS)

tamaño matriz	síncrono	Asíncrono
6000x6000	0.6367	0.3302
20000x20000	6.2661	3.2516
25000x25000	10.2213	5.3099

tamaño de problema.

En la tabla III mostramos los tiempos de ejecución totales de la versión de referencia y de la versión con transferencias y comunicaciones asíncronas. En la versión síncrona de referencia, los tiempos de computo y comunicación se acumulan en el total, ya que las operaciones se secuencializan. En la versión asíncrona, la mayor parte de los tiempos de comunicación se solapan con la computación, observandose tiempos totales que tienen apenas un incremento de entre un 1% y un 2% sobre los tiempos de comunicación de la versión de referencia.

C. Buffers vs. cudaMemcpy2D

En este trabajo comparamos dos aproximaciones para transferir los datos de los bordes verticales entre el host y el dispositivo y viceversa. Una con programación manual, ejecutando operaciones de marshalling y unmarshalling en buffers con datos contiguos. La otra utilizando directamente la función *cudaMemcpy2D* con los parámetros adecuados. En la tabla IV mostramos una comparación de tiempos de ejecución totales entre la cada una de las dos versiones. Observamos que entre las dos técnicas aplicadas, el uso de cudaMemcpy2D nos da un ligera mejora sobre el uso de buffers, ya que este método tiene diferentes optimicaciones a nivel del driver de CUDA. Además es mucho más sencillo que programar manualmente la operaciones sobre los buffers, y al ser parte del API de CUDA es más portable con arquitecturas futuras.

TABLA IV
TABLA CON TIEMPO DE COMUNICACIÓN DE BUFFERS VS. CUDAMEMPCY2D (EN SEGUNDOS)

tamaño matriz	buffer	cudaMemcpy2D
6000x6000	0.3268	0.3255
20000x20000	3.1919	3.1484
25000x25000	5.1980	5.1371

D. Escalabilidad de comunicaciones MPI

En esta parte del trabajo experimental estudiamos la escalabilidad de las comunicaciones asíncronas en MPI, ejecutando la parte computacional en los cores de la CPU. Ejecutamos dos tipos de experimentos. El primero para medir la escabilidad débil (*weak scaling*), aumentando el tamaño del problema proporcionalmente al número de dispositivos involucrados. Así podemos observar el rendimiento de la aplicacion cuando se amplia el numero de nodos distribuidos, con una carga constante para cada nodo. El segundo tipo de experimentos está orientado a medir la escalabilidad para un tamaño de problema fijo (*strong scaling*). Así podemos observar el comportamiento de la apliación con una carga total constante, mientras incrementamos el número de nodos usados. Los tiempos de computo se reducen, ya que al haber más procesos involucrados le toca una parte más pequeña de las estructuras de datos a cada uno. El impacto y efectos de los tiempos de comunicación se hacen más evidentes,

Tamaños de problema para este caso???

En el caso del clúster Trasgo, empezamos usando cores de una única máquina, incrementando el numero de cores de cuatro en cuatro, hasta completar su capacidad. Se continuan introduciendo cores de cuatro en cuatro de las siguientes máquinas, siempre completándo su capacidad antes de añadir la siguiente. El orden de introducción de las máquinas se ha escogido por su cantidad de cores, quedando establecido el orden siguiente: chimera, hydra, phoenix y thunderbird. En el caso final se están utilizando los 24 cores de chimera, los 12 de hydra, los 4 de phoenix y de thunderbird. En el caso del clúster CETA procedemos de manera similar rellenando máquinas con bloques de 4 procesos. Al ser máquinas homogeneas el orden en que se añaden los nodos no es relevante.

Tanto en la figura 5, como en la figura 6, vemos que la aplicación incrementa el tiempo cuando incrementamos los procesos en un mismo nodo, debido a que con pocos procesos, nos encontramos en una situacion de alta afinidad, donde los cores que se estan ejecutando se encuentran en un mismo nodo NUMA (misma CPU) y los movimientos de memoria y las comunicaciones entre los procesos son muy rápidas. Al incrementar el número de procesos, se introducen procesos en otras partes de la jerarquía, incrementándose los costes de comunicación. En el momento en que se empiezan a introducir procesos en otros nodos de la red, los costes de comunicación son ya similares. Por lo que la escalabilidad débil se mantiene estable, con un ligerísimo incremento hacia el final en el caso del clúster Trasgo al inroducirse tencnología de red más limitada (Ethernet 1Mb).

En la segunda parte del estudio comprobamos la escalabilidad para un tamaño de problema fijo (*strong scaling*). En la figura 7 se muestran los resul-

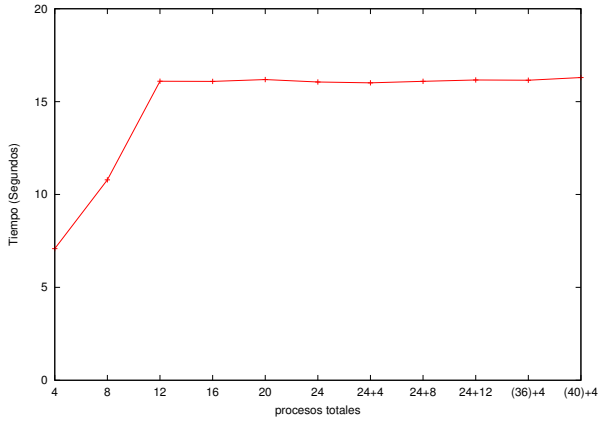

Fig. 5. Clúster Trasgo: Weak scaling con número de procesadores

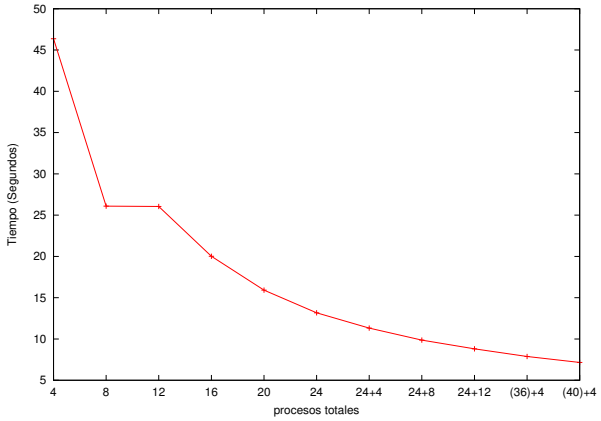

Fig. 7. Clúster Trasgo: Strong scaling con número de procesadores

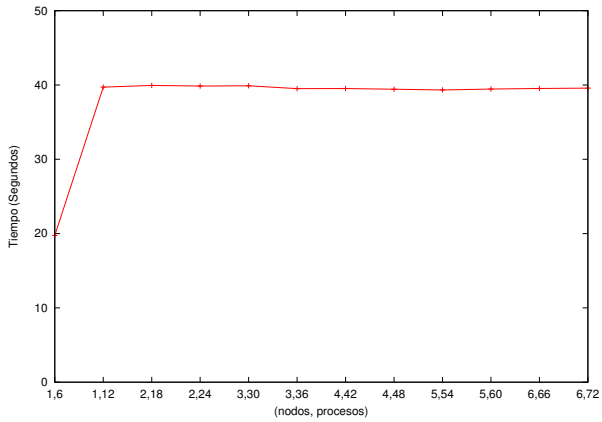

Fig. 6. Clúster CETA: Weak scaling con número de procesadores

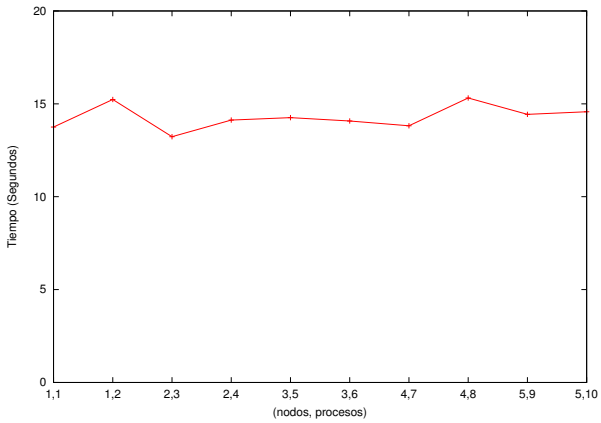

Fig. 8. Clúster CETA: Weak scaling con GPUs en nodos distribuidos

tados para el clúster Trasgo. Podemos ver la mejora clara en los tiempos del programa a medida que incrementamos el número de nodos que usamos, ya que la carga per nodo que realizamos es cada vez menor. Incluso con una tecnología de red basad en bus, como es Ethernet, la cantidad y frecuencia de las comunicaciones de este tipo de aplicaciones no llega a ser suficiente Las irregularides de la curva en los primeros puntos se debe de nuevo al cambio brusco en los costes de comunicación dentro de la jerarquía NUMA del mismo nodo.

E. Multi-GPU distribuidas

En esta parte del estudio nos centramos en estudiar la escalabilidad, tanto débil como fuerte (*weak, strong scaling*) cuando el cómputo se realiza con los dispositivos GPU, reduciendose los tiempos de cómputo e incrmentándose los de comunicación debido a la introducción de las transferencias entre la jerarquía de memoria del dispositivo y el host.

Tamaños de problema para este caso???

En la figura 8 mostramos los resultados de escalabilidad débil para el clúster CETA. Podemos observar que aparecen más irregularidades debidas a los efectos estocásticos de la red. Sin embargo, la escalabilidad se mantiene gracias a la capacidad de la tecnología de red de este clúster (Infiniband).

En la figura 9 mostramos los resultados de escalabilidad fuerte en el cluster Trasgo. Las primeras GPUs que se introducen son las de hydra (4 en el mismo nodo). Luego se van añadiendo GPUs de los otros nodos, introduciendo al final las de los nodos conectados con la tecnología de red más limitada. En este caso el tiempo de cómputo se reduce drásticamente gracias al uso de las GPUs, solapándose por completo con el tiempo de comunicación que domina el tiempo total. Los tiempos en los últimos puntos de la gráfica se ven afectados por la capacidad de la tecnología de red más limitada de los últimos nodos introducidos (Ethernet 1Mb).

Conclusiones

En este trabajo se han repasado y revisado técnicas de implementación eficiente de computaciones de tipo stencil para la ocultación de las latencias de comunicación entre host y GPU, y entre nodos, en un clúster de GPUs distribuido. Se han revisado diferentes optimizaciones y usos del API de CUDA. Los resultados muestran que la aplicación de técnicas sencillas de programación puede conseguir minimizar el impacto de las comunicaciones incluso en el caso de utilizar tecnologías de red de medio compartido de bajo coste para grados de escalabilidad acotados. Los resultados son prometedores en cuanto a la posi-

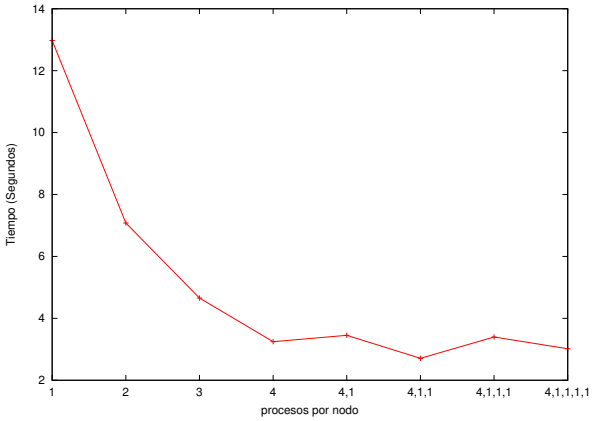

Fig. 9. Clúster Trasgo: Strong scaling con GPUs en nodos distribuidos

bilidad de resolver eficientemente problemas basados en stencils en grandes clústers con nodos dotados con aceleradores hardware y tecnolgía de red escalable.

El trabajo futuro incluye el estudio de aplicaciones stencil más complejas, otras técnicas no contempladas en el estudio, ajustar la partición de datos a la heterogenidad de los dispositivos y nodos, y la integración de las técnicas estudiadas en abstracciones de programación de sistemas heterogéneos de más alto nivel.

Referencias

[1] Einwg Lusk William Gropp, *Using MPI. Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 3rd edition, 2015.

[2] Brian Hamilton, Craig J. Webb, Alan Gray, and Stefan Bilbao, "Large stencil operations for GPU-based 3-D acoustics simulations," in *18th Int. Conference on Digital Audio Effects (DAFx-15)*, 2015.

[3] Paulius Micikevicius, "3d finite difference computation on gpus using cuda," in *GPGPU2*. March 2009, ACM Press.

[4] Mohammed Sourouri, Johannes Langguth, Filippo Spiga, Scott B. Baden, and Xing Cai, "CPU+GPU programming of stencil computations for resource-efficient use of GPU clusters," in *IEEE 18th International Conference on Computational Science and Engineering*. 2015, IEEE.

[5] Kaushik Datta, Samuel Williams, Vasily Volkov2, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick, "Auto-tuning the 27-point stencil for multicore," in *International Workshop on Automatic Performance Tuning (iWAPT)*, 2009.

[6] David Kirk and Wen mei Hwu, *Programming Massively Parallel Processors*, Morgan Kaufman, 2nd edition, 2013.