Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática

Ultraescalabilidad en sistemas heterogéneos: Integración de técnicas de gestión del paralelismo

Autor:
D. Senmao Ji Ye

Tutor:
Dr. Arturo González-Escribano

Resumen

El incremento en el uso de aplicaciones paralelas de gran escala dentro del cómputo científico, ha provocado el gran desarrollo de los último años de la programación paralela. Esto ha incrementado la complejidad en el desarrollo de este tipo de programación, con un número cada vez mayor de arquitecturas diferentes y tipos de aceleradores. Por otro lado, las restricciones físicas de los materiales y las limitaciones energéticas, obligan a optimizar el rendimiento de estas aplicaciones en orden de permitir una mayor potencia de cálculo.

Este trabajo presenta el estudio y desarrollo de un conjunto de técnicas para la mejora del rendimiento de la programación paralela, y facilitar su aplicación en sistemas heterogéneos. Busca ofrecer una solución diferente a las distintas vias de investigación que existen en la actualidad, combinando técnicas básicas para la aumentar la eficiencia de la programación paralela, con varias librerías de abstración, para desarrollar un modelo simple y óptimo de programación en paralelo en sistemas heterogéneos. Durante el desarrollo de este proyecto se estudiado, documentado y experimentado distintas técnicas de optimización de programación en CUDA y MPI, con el objetivo de obtener la mejor combinación de técnicas con el mayor rendimiento posible en aplicaciónes paralelas. Además se añadira un nivel de abstración superior para igualar la programación en sistemas heterogéneos.

Abstract

The increase in the use of large-scale parallel applications inside the center, in the development of the great development of the last times in the programming of the parallel. This has increased the complexity in the development of this type of programming, with an increasing number of digital architectures and types of accelerators. On the other hand, the physical restrictions of the materials and the energy constraints, force to optimize the performance of these applications in order to improve the computation. This work presents the study and the development of a set of technique for the Improvement of the performance of programming in the parallel, and easier its application on heterogeneous systems. It looks for offer a different point of view from the different routes of investigation on which exist today, combining the economies to increase the efficiency of the parallel programming, with several bookstores in the abstraction, to develop a simple and optimal model of programming in Parallel In heterogeneous systems. During the development of this project, several programming optimization techniques were studied, documented and experimented with in CUDA and MPI, in order to obtain the best combination of techniques for the greatest performance in parallel applications. Also as a level of abstraction at the top for programming in heterogeneous systems.

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

1.1. Contexto y motivación

La programación paralela ha sido durante los últimos años, un elemento esencial en aquellos campos ajenos al desarrollo informático donde se desarrolla y trabaja con programas con cargas de datos cada vez mayores, acercando la llegada a la denominada *computación exaescalar*, donde se requerira trabajar con cómputo de tamaños de exaFlops, es decir, mas de 10^{18} cálculos por segundo. Estas actividades se desarrollan normalmente en clústers con gran número de nodos, donde gracias a la conexión de múltiples equipos, podemos obtener una capacidad mucho mayor de cómputo, creado asi los supercomputadores. El supercomputador existente más potente del mundo durante la redacción de este artículo es el supercomputador chino *Sunway TaihuLight*, con mas de diez millones de cores y un consumo eléctrico de 15 megavatios, aunque múltiples potencias economicas del mundo, tienen grandes proyectos para sobrepasar la capacidad de cómputo del *Sunway TaihuLight* en un corto plazo, alcanzando la computación exaescalar en 2020. Esta gran demanda de capacidad de cómputo y una necesidad creciente en la reducción del consumo eléctrico, hace que la programación paralela sea unos de los campos en investigación informática más importantes de la actualidad.

Por otra parte, el desarrollo en el campo de la programación paralela, ha provocado la aparición de nuevos aceleradores, entre ellas la Xeon PHI de Intel, que combinada con las ya existentes GPUs de Nvidia, ha permitido una mayor diversidad y libertad en el desarrollo de estas aplicaciones, pero a su vez dificulta la adaptación de nuevas aplicaciones en sistemas con clústers heterogéneos donde se utilicen distintos aceleradores, ya que la programación de cada acelerador no se realiza de la misma manera. El programador debe realizar un estudio profundo de las particulalidades de los datos que se necesitan computar en cada momento, sobre las diferentes plataformas, y los distintos detalles arquitectónicos, para obtener un rendimiento óptimo de la aplicación. Por lo tanto, una de las lineas de investigación mas importantes, junto al de la optimización de cómputo, es el desarrollo de un modelo que permita una mayor adaptabilidad/portabilidad de una aplicación paralela a través de las diferentes plataformas.

La principal motivación del trabajo es el interés personal en el desarrollo y estandarización de la programación paralela. Durante la docencia recibida en el estudio del grado de Ingeniería Informática, se impartio una introdución a la programacion paralela y en particular a tres modelos de programación, OMP, MPI y CUDA. Durante mi estudio de esta asígnatura, conocí la poca estandarización que existe en cuanto a la optimización de código paralelo, aun contando detras de ellas con grande empresas especializadas como Nvidia. El uso de herramientas de tanto potencial y tan demandada como es la programación paralela y la poca estandarización de un modelo de progración paralela de alto rendimiento, implicaría una mayor dificultad en el desarrollo inicial de un programador novel. Esto me llevo a interesarme por desarrollar un modelo de óptimizacion de programación paralela en CUDA, buscando crear un modelo con un fácil aprendizaje inicial, usando llamadas a funciones existentes en la API de CUDA, pero inclu-

yendo otras características más complejas y potentes, como la mejora de portabilidad de un programa y su uso en sistemas heterogéneos mediante el uso de librerias de abstracción, para programadores más avanzados.

1.2. Planteamiento del problema

El foco de este trabajo es la programación de aplicaciones en clústers heterogéneos distribuidos. Estos clústers están compuestos por diferentes máquinas (nodos) interconectadas por red, en las que en cada una hay uno o más dispositivos aceleradores. Los dispositivos aceleradores son sistemas de cómputo completos (con procesadores y memoria propios) que se añaden al sistema principal.

Para conseguir aplicaciones que escalen adecuadamente en este tipo de sistemas, es fundamental que las latencias (tiempos de espera) relacionados con las operaciones de movimiento de datos entre los diferentes dispositivos implicados y a través de la red, se oculten, realizandolas de forma asíncrona mientras los procesadores computan otras partes del cómputo global que no necesite los datos que se están transfiriendo.

Aunque existen sistemas aceleradores integrados en el mismo chip de la CPU, en este trabajo nos vamos a focalizar en los sistemas que se conectan a un bus de expansión, como puede ser el bus PCI. Es la forma más genérica de conectar aceleradores especialmente los sistemas de cómputo masivos de gran potencia, como las tarjetas gráficas (GPUs) de alta gama, que aparecen sistemáticamente en los grandes supercomputadores. Estos aceleradores se controlan añadiendo al programa principal que se ejecuta en la CPU llamadas específicas para menejarlos. Al tener espacios de memoria diferentes, necesitan que el programa transfiera datos a la memoria del dispositivo acelerador antes de lanzar en el mismo rutinas de cómputo (kernels), y transferir los resultado de vuelta desde el acelerador a la memoria del host. Estas operaciones necesarias para manejar el acelerador reciben el nombre genérico de operaciones de *offloading*. Las operaciones de transferencia de datos, al realizarse a través de buses externos, son mucho más costosas que los accesos directos a memoria. Por tanto, es importante conseguir que estas operaciones, así como las transferencias a través de la red, sean asíncronas y solapadas con el cómputo.

Para conseguir aplicaciones con un alto nivel de escalabilidad en este tipo de sistemas heterogéneos el programador debe comprender y dominar los modelos de programación adecuados para los aceleradores y los modelos de intercambio de datos entre máquinas distribuidas o interconectadas por red, manejando adecuadamente las diferentes capas de comunicaciones o transferencias de datos para solapar las latencias de dichas transferencias con el cómputo.

En este trabajo estudiamos el caso práctico de la aplicación de este tipo soluciones para clústers con aceleradores GPU de Nvidia, que se manejan con el modelo de programación CUDA, utilizando sistemas de paso de mensajes MPI para comunicar las máquinas del clúster. Se estudiaran las posibilidades técnicas que nos ofrecen tanto CUDA como MPI para conseguir la escalabilidad deseada, y su posible integración en un modelo de programación mÃ¡s abstracto que las haga más transparentes al programador.

1.3. Solución propuesta

El objetivo principal de este trabajo es la documentación, investigación y desarrollo de optimicaciones software para mejorar el rendimiendo en la cómputación de aplicaciónes paralelas mediante el uso óptimo de llamadas nativas de CUDA y MPI, y la integración de estas en sistemas heterogéneos mediante una serie de librerias de abstración llamadas Hitmap [3] y Controladores [4], ofreciendo una modelo alternativo y diferente a los ya existentes. Para ello: (1) realizar un estudio profundo de tecnologías relacionadas con la comunicación asíncrona en transferencias de memoria entre host y GPU en CUDA; (2) Definir casos

de estudio para analizar el efecto de aplicar las diferentes técnicas estudiadas; (3) Integrar las técnicas estudiadas en el modelo de programación de Controladores; (4) Estudiar experimentalmente el efecto de aplicar las técnicas a los casos de estudio, y las ventaja o inconvenientes de utilizar el modelo de Controladores respecto a las implementaciones manuales.

Los objetivos de este trabajo son:

- Expandir mi conocimiento en el campo de la computación paralela para realizar un trabajo detallado y completo.
- Obtener rendimiento óptimo en las transmisiones de memoria de CUDA mediante el solapamiento de computación y comunicación.
- Obtener un modelo más abstracto de programación mediante el uso de Controladores para facilitar su programación y portabilidad en sistemas heterogéneos.

1.4. Definiciones y acrónimos

A continuación se enumeran los acrónimos o definiciones que se mostraran a lo largo del documento:

- **Clúster:** Conjunto de ordenadores conectados entre si, con el objetivo de formar una única computadora.
- **HPC:** High Perfomance Computing.
- **GPU:** Graphics Processing Unit. Es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante.
- **API:** Application Programming Interface. Es un conjunto de subrutinas, funciones y procedimientos que ofrece una biblioteca para ser utilizado por otro software como una capa de abstracción.
- **Kernel:** Dentro del campo de la computación paralela, recibe el nombre de kernel aquellas funciones que se realizan con el modelo de programación de CUDA.
- **SM:** Streaming Multiprocessor.

1.5. Estructura del documento

El documento se estructura de la siguiente manera: El capítulo dos presenta el estado del arte, donde se describira la situación actual del contexto de trabajo, vease las diferentes vias de investigación exitentes, y las diferencias que tienen estas respecto a nuestro trabajo. El capítulo tres contiene una descripción de la solución propuesta, de forma conceptual y técnica, ademas es una breve introducción al estudio experimental. El capítulo cuatro muestra el estudio experimental realizado, los resultado obtenidos, las plataformas utilizadas, los problemas elegidos y una síntesis de estos resultados. El capítulo cinco contiene las conclusiones obtenidas de este trabajo, además del trabajo futuro que nos permite este trabajo.

Capítulo 2

Estado del arte

En este capítulo se mostrará un resumen del estado de la investigación de la programación paralela en la actualidad, abarcando articulos sobre el manejo de clúster hasta el tratamiento de datos. Esta sección estara divida en dos apartados principales, donde la primera tratará sobre las tecnologías más relevantes que se utilizan en este trabajo y la segunda parte tratará de los trabajos que han sido realizados sobre el estado del arte.

2.1. Tecnologías relevantes

Durante el desarrollo de este proyecto se ha trabajado con una serie de tecnologías, cuya conocimiento es esencial para una correcta comprensión del trabajo. A continuación se procede a explicar de forma rapida y detallada las herramientas más importante.

2.1.1. MPI

MPI ("Message Passing Interface", Interfaz de Paso de Mensajes) es una API de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores, siendo el estándar para la comunicación entre los nodos que ejecutan un programa en un sistema de memoria distribuida La principal característica de esta API es que no precisa de memoria compartida, por lo que es ideal para trabajar en entornos de trabajo distribuidos.

Características

- **Estandarización:** MPI es la única libreria de paso de mensajes considerada como estándar, siendo soportado en la gran mayoria de plataformas de HPC.
- **Portabilidad:** Para el paso de una plataforma a otra, se necesita una pequeña o nula modificación de las aplicaciones.
- **Capacidad de mejora de rendimiento:** Aunque las implementaciones de los proveedores deberian ser capaz de explotar las características del hardware nativo para un rendimiento óptimo, MPI permite el desarrollo e implementación de algoritmos para un mayor rendimiento en apliaciones más específicas.
- **Funcionalidad:** Existen más de 500 rutinas definidas en MPI, ofreciendo una gran variedad de funciones para el desarrollo del programador.
- **Disponibilidad:** Existen multitud de versiones de implementación de MPI, tanto de dominio público como privado. Las más destacadas son MPICH, MVAPICH y Open-MPI.

Funcionamiento

MPI usa objetos llamados comunicadores y grupos para definir una colección de procesos que puedan comunicarse entre ellos. Dentro de cada comunicador, cada proceso recibe un número único asignado por el sistema cuando se inicia el proceso. Este número permite al programador espeficicar la fuente y el destino de los mensajes, y el uso de condiciones de control para la ejecución del programa.

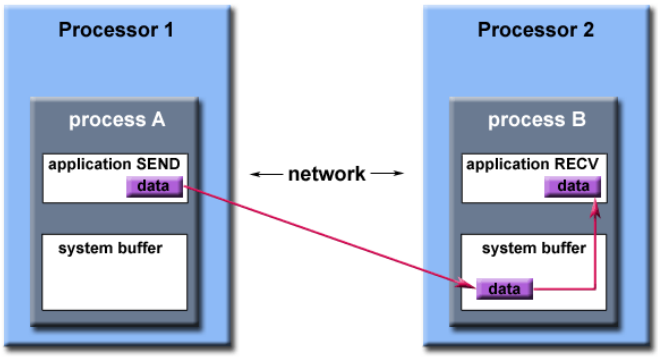

Figura 2.1: Envio de mensajes en MPI [1]

El paso de mensajes puede realizarse tanto de forma síncrona como asíncrona. Para las comunicaciones asíncronas, MPI permite el envio de múltiples mensajes sin la necesidad de que se recibas los anteriores, mediante un sistemas de buzones donde se almacenan los mensajes pendientes, a la espera de que un proceso los reciba.

MPI controla el orden de envio y llegada de los mensajes, garantizando que los mensajes no se superaran unos a otros.

Para realizar un programa básico con MPI, nos basta con utilizar el paso de mensajes pareado, pero MPI nos ofrece una serie de potentes rutinas de comunicación colectiva. Estas pueden ser para un fin de sincronización de procesos, movimiento de datos, o computación colectiva. Estas rutinas deben ser realizadas por todos procesos que se encuentren por defecto en el alcance del comunicador, pudiendo el programador añadir más procesos de forma adicional. Si un solo de los procesos no alcanza esta rutina, provocara un fallo en del programa. Es responsabilidad del programador asegurar que todos los procesos sean capaces de alcanzar la rutina en tiempo de ejecución.

2.1.2. CUDA

CUDA es una plataforma de computación paralela y modelo API creado por Nvidia. Nos permite un incremento drástico en el rendimiento de cómputo usando cualidades de las GPUs, ofreciendo al programador un acceso directo al conjunto de instrucciones virtuales de la GPU y los elementos de cómputo paralelo que este dispone para la ejecución de kernels.

El uso de GPUs para el uso cotidiano, como el cómputo gráfico en videojuegos, recibe el nombre de GPGPU (General-Purpose computing on Graphics Processing Units)

Arquitectura CUDA

La arquitectura actual de CUDA consiste en una serie de componentes, resaltados en verde en la siguiente figura:

1. Motores de computo paralelo dentro de las GPU de Nvidia

2. Soporte a nivel de kernel del sistema operativo para la inicialización, configuración, etc.

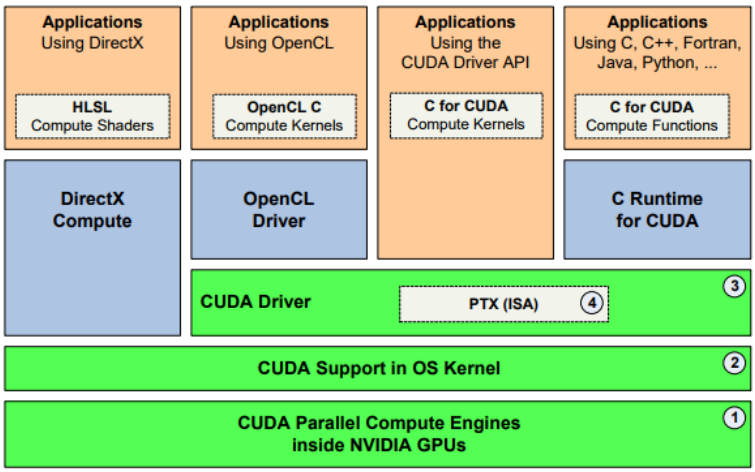

Figura 2.2: Arquitectura de CUDA [2]

3. Controlador del modo usuario, que proporciona una API a nivel de dispositivo para los desarrolladores.

4. Arquitectura de conjunto de instrucciones PTX (ISA), Parallel Thread eXecution (Industry Standard Architecture), para kernels y funciones de cálculo paralelo.

Arquitectura GPU

La estructura de la GPU esta formada por dos componentes principales.

- Memoria Global.
  - Es el equivalente device a la RAM en un servidor CPU.
  - Accesible desde la CPU y la GPU.
  - Los últimos modelos producidos por Nvidia alcanzan los 12 GB.
  - Ancho de banda de hasta 300 GB/s.
- Streaming Multiprocessors (SM) (Ver figura 2.3).

  Son los componentes individuales más básicos diseñados por Nvidia para realizar el procesamiento de cómputo.

  Cada SM contiene:

  - Miles de registros que pueden ser particionados entre procesos en ejecución.
  - múltiples estructuras de Cache:
    - Memoria compartida para el intercambio rápido de datos entre hilos.
    - Caché constante para la difusión rapida de lectura de memoria constante.
    - Caché de textura para agregar el ancho de banda de la memoria de textura.
    - caché L1 para reducir la latencia a la memoria local o global.
  - Planificadores de *warp* que pueden cambiar rapidamente de contexto entre hilos y emitir instrucciones a los warp que estan preparados para la ejecución
  - Nucleos de ejecución para operaciones de enteros y de punto flotante.
    - Operaciones de enteros y de punto flotante de precisión simple.

- Punto flotante de doble precisión.
- Unidades de funciones especiales (SFU) para funciones transcendentales de punto flotante de simple precisión.

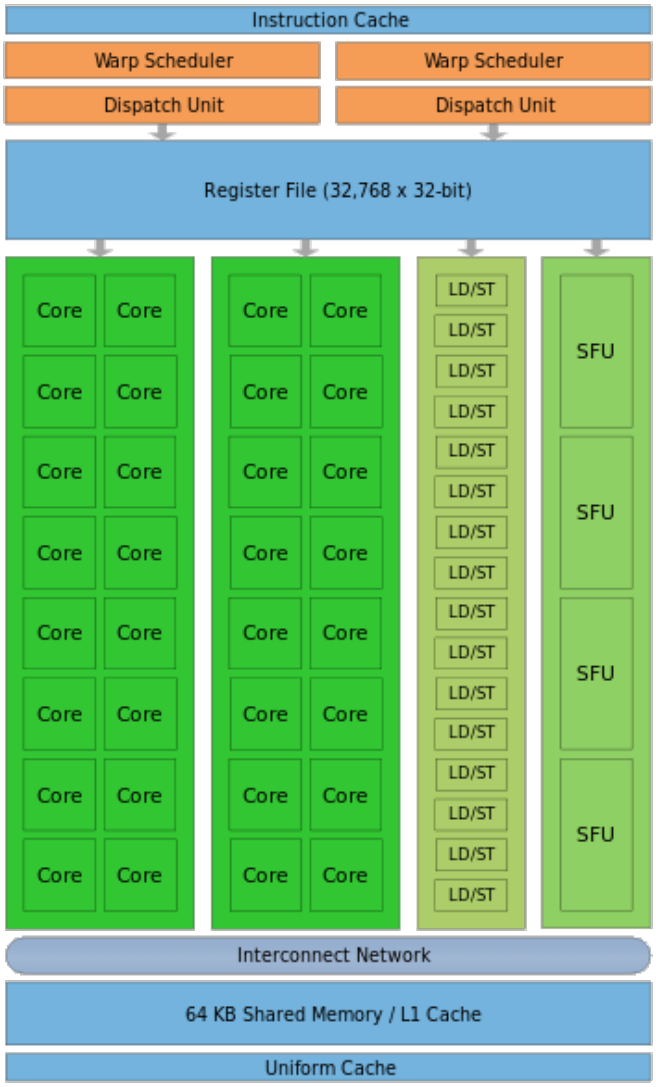

Figura 2.3: Arquitectura de un SM de las GPU Fermi [2]

Modelo de programación

En la actualidad existen dos APIs de CUDA, CUDA driver API y CUDA runtime API. Aunque ambas funcionalmente son similares, CUDA runtime es la más utilizada por tener un mayor nivel de abstración.

Cuda runtime facilita el manejo de código device proveyendo al programador de inicializaciones implicidas, manejo de contexto y administración de modulos. El código host en C generado por nvcc, el compilador de CUDA, esta basado en CUDA runtime, por lo que las aplicaciones que enlazan este código deben utilizar la versión de runtime.

En contraste, CUDA driver API requiere una mayor cantidad de código, dificultando su programación y su posterior depuración, pero ofrece un mayor nivel de control y es independiente del lenguaje.

Kernels CUDA

Cuda permite al programador definir unas funciones en C llamadas kernel, que cuando son invocadas, estas se ejecutan en paralelo con el valor de un número de hilos que hayamos pasado como parámetro. Un kernel se define mediante la declaración *__global__*, y el número de hilos que queremos ejecutar para este kernel se especifica en un nuevo campo de configuración de ejecución, que se identifica con «¡... »¿. Este contiene cuatro campos:

- **Dg** (dim3) especifica el numero de dimensiones y el tamaño del grid.
- **Db** (dim3) especifica el numero de dimensiones y el tamaño de bloque.
- **Ns** (size_t) especifica el numero de bytes en memoria compartida que es reservada dinamicamente por bloque.
- **S** (cudaStream_t) especifica el stream asociado a la ejecución del kernel. Este parámetro sirve para realziar llamadas asíncronas.

Transferencia de memoria

El modelo de programación CUDa asume un sistema compuesto de un host y un device, cada uno con su propia memoria separada. Los kernels operan sobre la memoria device, por lo que provee de funciones para realizar las reservas de memoria, liberación, copia de datos en device, y la transferencia de datos entre host y device.

La memoria de device puede reservarse tanto como memoria linear o como CUDA arrays, una estructura de datos optimizada para el tratamiendo de matrices.

La forma más común de realizar las reservas es memoria es de forma linear, mediante la llamada a *cudaMalloc()* y liberarla con *free()*. De igual manera el tranferencia de datos más típica es realizada con memoria linear, mediante la llamada a cudaMemcpy(). La estructura de esta llamada es muy simple, configurandola mediante cuatro parámetros:

- **dst** (void *) dirección de memoria de destino.
- **src** (const void *) dirección de memoria de la fuente.
- **count** (size_t) tamaño de datos que se quiere transferir en bytes.
- **kind** (enum cudaMemcpyKind) tipo de transferencia, que pueden ser de tipo Host to Host, Host to Device, Device To Host, Device to Device, o por defecto.

Estructura de un programa básico

En la figura 2.4 podemos observar el diagrama de flujo más básico de una aplicación CUDA.

Previamente a este proceso, se debe realizar las reservas de memoria correspondientes, tanto en la memoria de Host como en la memoria de Device.

1. Se realizar es la copia de datos que se quieren computar de forma paralela en el kernel desde la memoria host (CPU) a la memoria de device (GPU) mediante **cudaMemcpy()**. También podemos encontrarno funciones como *CudaMemcpy2D* o *CudaMemcpyAsync*, que nos permiten la copia de estructura en dos dimensiones y realizar la transferencia de datos de forma asíncrona, respectivamente.

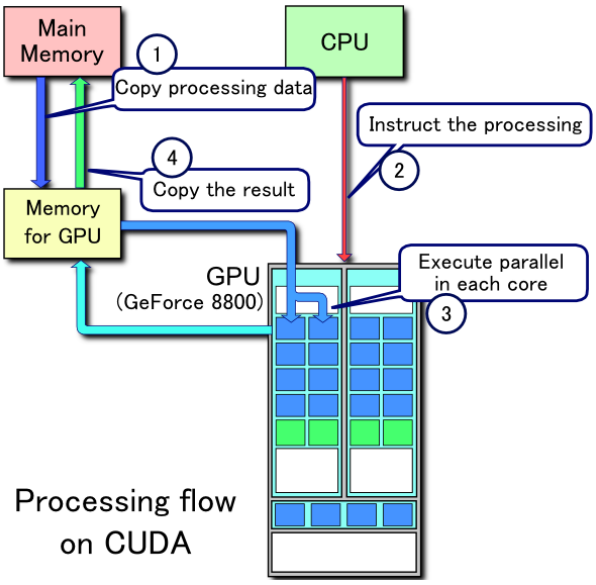

Figura 2.4: Mapa de flujo proceso en CUDA [2]

2. Se realiza la llamada al kernel que se haya implementado. Durante esta llamada se pasan los parámetros a la función, incluyendo ademas el número de hilos y bloques que queremos que se ejecuten.

3. El kernel invoca a todos los hilos y procesa los datos de forma paralela, guardando los resultados en la memoria de device. Mediante una llamda asíncrona al kernel, se puede conseguir solapar este caso con el paso anterior.

4. Por último, una vez finalizado el kernel, copiamos los datos de la memoria device con los resultados, a la memoria host.

2.1.3. Hitmap

Hitmap[3] es una librería de alta eficiencia para la una jerarquía de "tiles", que es una estructura de datos compuesta de un vector y una serie de metadatos, el mapeo de matrices y estructuras dispersas. Esta diseñado ara ser usado en lenguajes de programación paralela de tipo SPMD (Single Process, Multiple Data), con el objetivo de simplificar la programación paralela, ofrecer funcionalidades para crear, manipular, distribuir y comunicar, tiles y jerarquia de tiles.

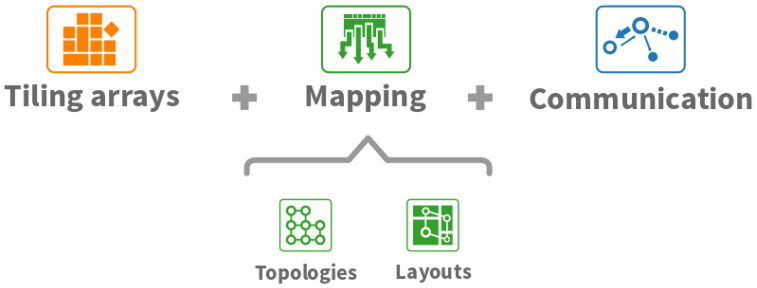

Figura 2.5: Componentes de la libreria Hitmap

Hitmap incorpora múltiples funciones para realizar la jerarquía de tiles y mapeo de matrices. Esta libreria esta diseñada para simplificar el uso global de la computación paralela. En Hitmap, el diseño de la estructura de datos, y las técnicas de equilibrio de carga son modulos independientes que pertenecen a

un sistema de modulos. Las funciones son invocadas desde el código y aplicadas en tiempo de ejecución cuando son necesarioas, usando información interna de la topología del sistema destino para la distribución de datos. El programador no necesita calcular el número de procesadores físicos. En su lugar, debe utilizar patrones de comunicación de alto nivel de abstracción para la distribución de los tiles en cualquier nivel de grano. Por lo tant, las operaciones de codificación y depuración con estructuras de datos completas son mucho más sencillas.

Estructura de Hitmap

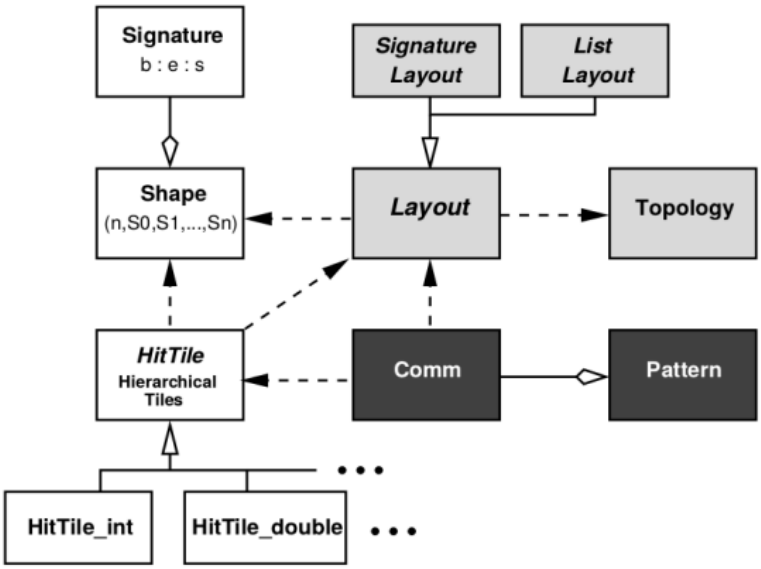

Figura 2.6: Estructura de Hitmap

Como podemos observar en la figura 2.6, Hitmap es una estructura compuesta de múltiples clases. Las cajas blancas son las clases que gestionan los dominios y las estructuras de datos (Funciones de particionado). Las cajas en tono gris claro, representa las clases que aplican la partición y mapeado del dominio (Funciones de distribución). Las cajas gris representan las clases que generan patrones de comunicación adaptables y portables (Funciones de comunicación).

- **Funciones de particionado (tiling):** En esta sección se encuentras aquellas clases que se encargan de definir y manipular los tiles y arrays. Se puede utilizar de manera independiente al resto de funcionalidades de Hitmap, para generar distribuciones de datos o optimizar la localidad en el código secuencial.

  - **Signature (HitSig):** Las signaturas con tuplas de tres números enteros, que represnetan un subespacio de índices de arrays en un dominio unidimensional. Estos tres enteros definen el índice inicial, el índice final y el salto que hay entre las celdas de la signatura. Un ejemplo de definición de signatura seria (0:15:2), que representa un array unidimensional de 8 elementos, con saltos de un elemento entre ellas.
  - **Shape (HitShape):** Es una estructura de representa un subespacio de índices de un array. Esta formado por un conjunto de n Signatures, definiendo un array n-dimensional.
  - **Tile (HitTile):** Es la estructura de datos similar al array, definida por un Shape y sus elementos tienen un tipo definido. Poseen un ordenación jerárquica en base al número de subselecciones que se han realizado sobre el Tile original, permitiendo un reparto, localicación y acceso más eficientes.

- **Funciones de distribución (mapping):** Esta sección agrupa las clases que distribuyen los datos y el métodos de diseño para particionar dominios de forma automática. Estos elementos están orientados a la distribución de datos y tareas en entornos paralelos.

  - **Topology:** Clase abstracta que permite crear topologías virtuales, ocultando los detalles de la topología física. Dependiendo de la topologia elegida, se definen qué procesadores están activos en tiempo de ejecución, desactivando aquellos que no son usados.
  - **Layout:** Permiten distribuir un Shape entre los procesadores de una topología. Esta potilica de partición aplicada depende del tipo de Layout elegido. El layout además almacena las relaciones de vecindad de unos procesos con otros, para poder hacer referencia a estas en caso de que sea necesaria en comunicaciones posteriores.

- **Funciones de comunicación:** Implementan la creación de patrones de comunicación reutilizables para tiles distribuidos. Estas funciones son una abstracción del modelo de paso de mensajes (MPI), para comunicar tiles entre procesadores virtuales.

  - **Communication:** Representan la información para síncronizar o comunicar tiles entre procesos. Permiten crear diferentas esquemas de comunicación, en término de dominios de tiles, información de los objetos del layout y reglas de los vecino de la topología.
  - **Pattern:** Es una composición de objetos *Communication*, que permiten realizar varias comunicaciones con una única llamada.

2.1.4. Controlador

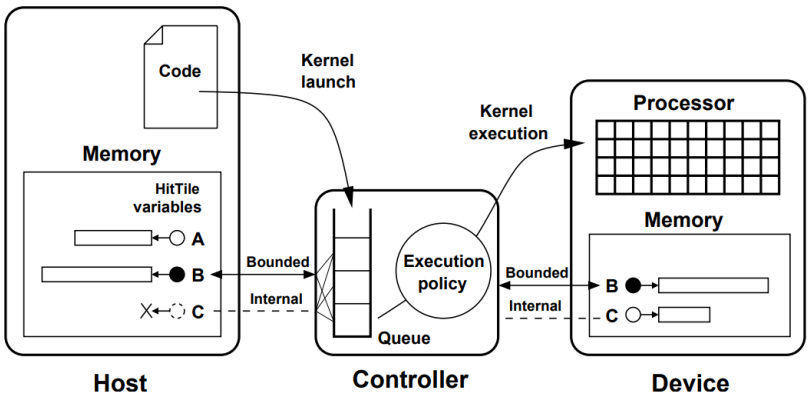

Figura 2.7: Estructura de un controlador

El grupo Trasgo desarrolló una entidad abstracta llamada Controlador[4], que permite al programador gestionar fácilmente las comunicaciones y los detalles de lanzamiento de kernels, sobre aceleradores hardware de forma transparente, ya sean coprocesadores XeonPhi o GPUs. Este modelo de abstracción nos ofrece la posibilidad de definir y lanzar kernels en procesadores multi-core con la misma abstración y metodología usada en los aceleradores. Combina internamente diferentes modelos de programación nativos y tecnologías para explotar el potencial de cada tipo de dispositivo. Además, el modelo también permite al programador simplicar la selección adecuada de parámetros para varios parámetros de configuración, que son seleccionados durante el lanzamiento del kernel. Esto se realiza a través de un proceso de caracterización cualitativa del código del kernel que se va a ejecutar.

Modelo del Controlador

El Controlador introduce una forma simplificada de programar aplicaciones que pueden ser explotadas en plataformas de computación heterogénea, que incluyan acceleradores y/o CPUs multi-core. El Controlador maneja la ejecución de una serie de kernels. Estos kernels declarados como funciones, son coordinados por las entidades de Controlador. Estas manejan automáticamente los dos conceptos usandos en un programa que busca sacar rendimiento a los aceleradores.

- Administración de kernels, incluyendo los de lanzamiento y configuración. El Controlador maneja el despleyado y ejecución de secuencias de kernels en el dispositivo computacional asociado al Controlador. El controlador puede incluir políticas de ordenación para mejorar la ténica de concurrencia de kernels o intercalar las comunicaciones con la computación.
- Administración de datos, incluyendo las transferencias de datos realizadas a través de las jerarquías de memoria del host y de los aceleradores, y la abstracción utilizada para acceder a los elementos de datos independientemente del dispositivo de destino, la indexación del espacio de los hilos o del diseño de datos.

Transmisiones de datos

El Controlador maneja de forma transparente las imagenes de las estructuras de datos del host, dentro de la memoria del acelerador. El Controlador decide cuando y como realizar las transferencia de datos, dependiendo de las estructuras de datos que se usan en los kernels preparados para el lanzamiento. El programador puede utilizar el mismo nombre para las variables del host o del la memoria del acelerador de forma indistinta, ya que el controlador se encargará de su selección.

Dependiendo de la función de las estructura de datos dentro del kernel, podemos distinguir dos tipos:

- **Variables enlazadas:** Son variables host que contienen una imagen en la memoria del acelerador. El controlador define una operación de enlazado del host al Controllador. Una vez que la variable es enlazada al Controlador, los datos de la variable en la memoria del host no debe ser modificada hasta que se ejecute la operación de desenlazado, que transmitira los datos de la memoria del device al host, si estos han sido modificados.
- **Variables internas:** Son variables cuyo alcance esta delimitado a la memoria del acelerador, por lo que nunca se realiza una transmisión de datos con el host de estas variables. Esta estructura de datos tiene como objetivo crear copias completas de una variable en la memoria del accelerador, permitiendo el uso de estas en los lanzamientos de kernel.

Declaración de kernels

Un kernel del modelo del Controlador se define utilizando la primitiva *KERNEL_¡type¿*, donde el valor *type* puede estar vacio, indicando el lanzamiento de un kernel genérico para cualquier tipo de acelerador, o especificando el tipo de dispositivo sobre el que se quiere realizar. Esto nos permite realizar distintas optimizaciones de un mismo kernel según el dispositivo que utilicemos. La versión actual de Controlador incluye las primitivas *KERNEL_GPU* para código CUDA con destino a GPUs de Nvidia, *KERNEL_CPU* para código máquina con destino a conjuntos de cores CPU, y *KERNEL_GPU_WRAPPER* para codigo máquina que incluye llamadas a librerias especializadas de GPU.

El paso de parámetros en los kernels se realiza mediante tuplas de información por cada parámetro. Esta tupla esta formada por el tipo, el nombre y su rol. Este rol puede ser de cuatro tipos distintos:

- **IN:** para parámetros HitTile de entrada, permitiendo solo la lectura.

- **OUT:** para parámetros HitTile de salida, permitiendo solo la escritura.
- **IO:** para parámetros HitTile de entrada y salida, permitiendo las operaciones de lectura y lectura.
- **INVAL:** para parámetros de entrada de cualquier tipo pasado por valor.

Caracterización de kernels

La caracterización del kernels permite al programador pasar al sistema los valores más adecuados para el lanzamiento del kernel, con cuatro parámetros:

1. El número de dimensiones del espacio de hilos. Puede tomar los valores enteros 1, 2 o 3.
2. Indicar la propiedad de coalescencia de los patroces de acceso a memoria global (full, medium o scatter).
3. El ratio de operaciones aritméticas y operaciones lógicas por acceso de memoria global (high, medium o low).
4. El ratio de acceso a memoria compartida e un bloque, por cada acceso a memoria global (high, medium, low).

```
/* Recurrence equation kernel */
KERNEL_CHAR(kRecurrence, 2, full, high, low)

/* Black Scholes kernel */
KERNEL_CHAR(kBlackScholes, 1, full, medium, low)

/* Stencil Jacobi kernels */
KERNEL_CHAR(kCopy, 2, full, low, low)
KERNEL_CHAR(kUpdate, 2, medium, medium, medium)

/* GPU matrix multiplication kernel */
KERNEL_CHAR(kMatrixMult, 2, fixed-square-32)
```

```
/* Kernel wrapper for cuBLAS matrix mult. */
KERNEL_GPU_WRAPPER( HitTile_float A,
                    HitTile_float B,
                    HitTile_float C ) {
  const float alpha = 1.0f;
  const float beta  = 0.0f;
  cublasHandle_t handle;

  cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
       B.dimx, A.dimx, A.dimy, &alpha,
       hit_ktileRawData(B), B.dimx,
       hit_ktileRawData(A), A.dimx, &beta,
       hit_ktileRawData(C), B.dimx );
}
```

Figura 2.8: Caracterización de un kernel con Controladores

Lanzamiento de kernels

La función *CtrlLaunch* prepara el lanzamiento de un kernel, con los parámetros correspondientes, al dispositivo de cómputo asociado al controlador. El lanzamiento del kernel sera puesto en la cola, y eventualmente ejecutado con la configuración definida durante la caracterización. La versión actual de Controlador, permite solo una política First-Coming-First-Service para la cola de kernels. El lanzamiento del kernel viene con los siguientes parámetros: (1) El Controlador asociado; (2) el nombre del kernel; (3) Índice del espacio del conjunto de hilos; (4) El número de parámetros que solicita el kernel y (5) los parámetros reales para la ejecución del kernel.

El índice del espacio del conjunto de hilos se define mediante una estructura *CtrlThreads*, que funciona de forma análoga al tipo *dim3*, almacenando el número de dimensiones y el tamaño de cada una.

2.2. Trabajos relacionados

En esta sección se mostrara una serie de trabajos de investigación que ofrecen una visión del problema relacionado con a nuestro trabajo, aunque aplicando técnicas notablemente distintas a las utilizadas en nuestro trabajo.

2.2.1. HPX

HPX [5] es un sistema de tiempo de ejecución de C++ de uso general, para aplicaciónes paralelas y distribuidas. Extiende el estandar de C++11/14 para facilitar operaciónes distribuidas, permitir paralelismo basado en restricciones de grano fino, y dar soporte en tiempo de ejecución de la gestión de recursos adaptativos. Esto proporciona una API que permite la programación, la composición y portabilidad de las aplicaciones paralelas de los usuarios.

Características

- **Prioriza la capacidad de ocultar latencias sobre la capacidad de reducirlas.** El objetio es aprovechar el tiempo de las latencias en la comunicación para realizar de forma simultanea trabajo útil, ocultando asi las latencias al usuario.

- **Adopta el uso de paralelismo de grano fino en lugar de hilos de carga pesada.** Esta se beneficia con el uso de colas de procesos, que mejoran su rendimiento cuanto menos sean el tamaño de los procesos.

- **Sincronizaciones basadas en variables:** Para evitar los costes que supone el uso de sincronizaciones globales, HPX introduce el concepto de *Futuro* en las variables. Esta permite agregar a las variables un valor extra, que indica si el valor de esa variable esta asignada, o aun esta pendiente de ser asignada en un futuro. Esto permite realizar una sincronicación en base a estas variables, por ejemplo, cuando son pasados como parámetros a una función.

- **Mejor mover el trabajo a los datos, en vez de pasa los datos al trabajo.** Muchos trabajos demuestran que por norma general, en numero de bytes necesarios para codificar un operación concreta, es mucho menos que el necesario para codificar los datos que utiliza la operación. Debido a que la mayoria de aplicaciones actuales utilizan MPI para el paso de mensajes, el cual es un modelo centrado en el paso de datos entre nodos, pocas son las aplicaciónes que toman esta alternativa.

- **Direcciones de memoria global uniformes.** MPI proporciona ninguna ayuda en la distribución de datos ni su localización, dejando el trabajo de descomponer los datos entre todos los nodos al programador. Creando una sola memoria global entre todos los nodos, nos permite tener un control de la localidad flexible y adaptable.

- **Computación lanzada por mensajes.** Siguiendo con la idea de dar una opción distita de MPI, HPX optar por el uso de tareas de cómputo mediante mensajes. Esta tiene como objetivo evitar el gasto que se producen en la recepción de mensajes que se producen en MPI, realizando envio de mensajes que lanzan tareas de cómputo por si mismas, que las procesara el sistema de cola de procesos.

Ventajas e inconvenientes

Esta herramienta ofrece una gran variedad de optimizaciones con el objetivo de optimizar la programación de aplicaciones paralelas en sistemas heterogéneos, bastantes distintas de las propuestas en este trabajo, que seran interesantes analizar personalmente en un futuro.

Aunque la mayoria de características que ofrece esta tienen un gran potencial de mejorar el rendimiento de una aplicación paralela, aun no esta claro su rendimiento total debido al sobrecoste producido, entre otras cosas, por el uso de las variables de tipo *Future*, que obligarian a realizar una gran cantidad de comunicaciones en sistemas distribuidos.

2.2.2. ARMCI-MPI

ARMCI-MPI [6]muestra una interesante herramienta para mejorar el uso de aplicaciónes en clústers. Motivados por el uso cada vez mas común de conjuntos de datos que superan el tamaño de un solo nodo, disenãn esta herramienta para proveer de un soporte adecuando a las *global array*, que son estructuras de datos que se reparten entre múltiples nodos, usando comunicaciones no pareadas de MPI.

Características

- **Traducción de direcciones y rangos.** ARMCI soporta dos modelos de creación de grupos de procesas: colectivo y no colectivo. La creación de grupos colectiva es implementando usando las llamadas nativas de MPI. Sin embargo, no es posible obtener una versión no colectiva usando la interfaz de creación de comunicadores de MPI. En cambio, utilizan la creación recursiva de intercomunicadores y un algoritmo de fusión desarrollados en un trabajo previo.
- **Redirección de memoria local a datos global.** Las operaciones de carga y almacenamiento de datos en una ventana MPI crea conflictos con todos los otros accesos a esa región. Una ventana MPI es la estructura de datos que utiliza MPI para las comunicaciones no pareadas, caracterizado por una memoria pública y una memoria local.

Ventajas e inconvenientes

El uso de memoria global en un cluster de nodos y el de ventanas MPI, implica que los acceso a una memoria local de un nodo han de sincronizarse posteriormente con la memoria pública de este, produciendo sobrecostes y perdidas en la eficiencia.

2.2.3. Optimizing MPI Communication on Multi-GPU Systems using CUDA Inter-Process Communication

En este artículo de investigación [7] se presentan una opción para mejorar las comunicaciones MPI en clusters con GPUs, mediante el uso de CUDA IPC (Inter-Process Communications). Este modelo soluciona el problema de la creación y mapeado los identificadores de memporia de CUDA IPC. En cuando a las comunicaciónes, ofrece una versión propia de comunicaciones pareadas y no pareadas.

Comuncaciones no pareadas

- Separa la comunicación de la sincronización.
- Utiliza la estructura de window o ventana.
- Añade una serie de funciones de comunicación y sincronización.
- Creación y mapeo de handles de memoria IPC durante la creación de las ventanas.
- Sincronicación mendiante eventos de CUDA.

Comunicaciones pareadas

Este modelo nos ofrece ciertas ventajas según el tamaño de las comunicaciones. Para comunicación de mensajes pequeños, minimiza los overheads de sincronización, usa memorias intermedias para la comunicación host-host y realiza la sincronización mediante eventos de cuda. Para las comunicaciones con gran tamaño de mensaje, minimiza el número de copias con un protocolo de citas y minimiza el gastos producidos en el mapeado de memoria, mediando el uso de mapeado cache.

2.2.4. libWater

Esta herramienta [8] utiliza OpenCL (Open Computing Language) como plataforma de desarrollo. OpenCL es un framework que consta de una interfaz de programaci'on de aplicaciones y de un lenguaje de programación. Juntas permiten crear aplicaciones con paralelismo a nivel de datos y de tareas que se puede ejecutar tanto en CPU como en GPU, por lo que esta orientado a la programación en plataformas heterogéneas.

libWater presenta un enfoque uniforme para la programación de sistemas de computación heterogéneos distribuidos. Se compone de una interfaz sencilla, compatible con el modelo de programación OpenCL y un sistema runtime que amplia las capacidades de OpenCL más allá de las plataformas simples y nodos de cálculos individuales. libWater mejora el sistema de eventos de OpenCL al habilitar la sincronicación inter-conexión e inter-nodo del dispositivo. Además el sistema de runtime utiliza los datos de dependencia reforzada por la sincronización de eventos, para construir diámicamente un DAG de comandos en cola, que permiten una clase de optimizaciones en tiempo de ejecución avanzadas.

Capítulo 3

Descripción de la solución

En esta sección se divide en tres apartados: La primera se compone de la descripción de los casos de estudio que se van a realizar; la segunda describe las técnicas de comunicación y optimización; la tercera describe detalladamente cómo se han aplicado estas técnicas a los casos de estudi; y la cuarta, cómo se integran estas técnicas en el modelo abstracto de Controladore [4].

3.1. Casos de estudio

3.1.1. Multiplicación de matrices

La multiplicación de matrices es un problema es muy corriente en un gran número de algoritmos en el álgebra lineal (Sistemas de ecuaciones, cálculo de estructuras, determinantes...).

Se caracteriza por ser un algoritmo muy simple, tanto conceptualmente como a la hora de implementar, por lo que hace que sea el benchmarking base de los estudios de programación paralela. El procedimiento es el siguiente: dado una multiplización de dos matrices M y N, y la matriz resultado R, cada valor de las celdas (m,n) de R esta compuesto por el producto de la fila m de la matriz M, y la columna n de la matriz N. Notese que el cálculo de cada valor de la matriz R, no tiene ninguna dependencia de datos entre ellas, por lo que pueden calcularse de forma simultanea.

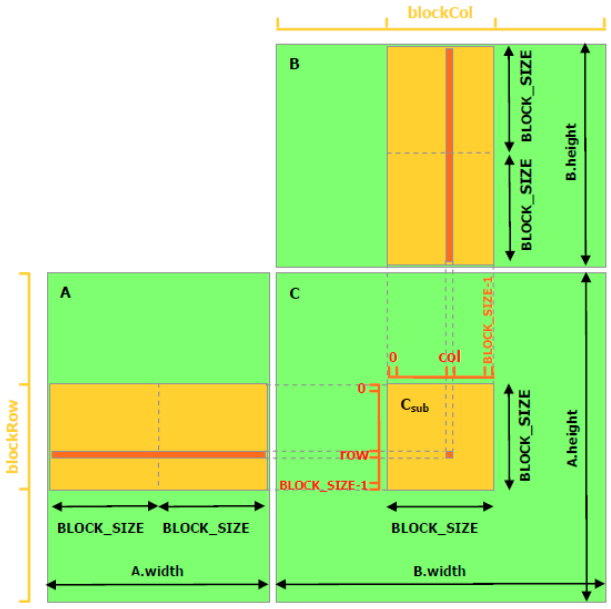

Figura 3.1: Multiplicación de matrices en cuda

La multiplicación de matrices son problemas fáciles desarrollar, que disponen de una gran cantidad de

cómputo (orden de N al cubo), y realizan un movimiento de datos considerable de toda la matriz. Este caso nos servira para probar las optimizaciones más genéricas que queremos evaluar.

3.1.2. 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. 3.2). 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.

```
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;
 }
}
```

Figura 3.2: Código de Jacobi en 2D

El patrón o stencil se puede complicar con celdas vecinas a mayor distancia (radio del stencil), de forma no equilibrada en las diferentes dimensiones o ejes [9], hasta llegar a métodos complejos con patrones que se alteran de forma dinámica mientras avanzan las iteraciones.

Este tipo de aplicaciones nos ofrece un comportamiento totalmente distinto al de la multiplicación de matrices, siendo programas con poco cómputo, la necesidad de sincronización entre vecinos, y un movimiento de datos pequeño por iteración.

3.2. Optimizaciones

A continuación se describirá por separado, cada una de las optimizaciones que se han estudiado y elegido en el desarrollo de este trabajo.

3.2.1. Solapamiento de cómputo y comunicación

La primera optimización que se estudia en este trabajo es la ejecución simultanea de trabajo de cómputo y transmisiones de datos, solapando los coste de ambas, de una aplicación paralela. Para poder aplicar esta técnica, debemos realizar un estudio previo del código y las dependencia de datos que existen

en ella. Los parámetros de entrada y de salida de un kernel no deben de tener ninguna dependencia de datos con las variables que se quieren transmitir de forma simultanea.

Existen múltiples formas para permitir el uso de esta técnica en la mayoria de programas, entre las que encontramos, el uso de una granuladidad más pequeña para reducir las dependencias, o el uso de estructuras de datos o variables auxiliares.

Más adelante aplicaremos la esta técnica en el caso de la multiplicación de matrices, que como veremos, nos aportara las mejoras en el rendimiento que esperabamos, pero a cambio de duplicar el eso de memoria. Esta técnica no sera eficaz en los casos donde se use gran parte de la memoria del sistema, aunque no se descarta su uso parcial o combinado con otras técnicas para seguir optando por esa mejora del rendimiento (Realizar un duplicado de media matriz, por ejemplo).

3.2.2. Tamaños de bloque

Durante los lanzamientos de los kernels de CUDA, se deben definir el tamaño de grid y de bloque que queremos ejecutar con el kernel, tal y como explicamos en el Capťulo 2. Una selección aleatoria o incorrecta en los tamaños grid y de bloque, puede afectar considerablemente en el rendimiendo del programa. Para que una aplicación funcione de forma óptima respecto al tamaño de bloque, como base, debemos utilizar tamaños de bloque que sean múltiplos de 32, que es el tamaño de warp, que es el conjunto mínimo ejecuciónes de datos que procesa en paralelo las SM *(Streaming Multiprocessor)*, que se encuentra en casi la totalidad del hardware disponible de Nvidia. Esto implica que, si al realizar trabajo computacional en las GPUs, no ajustamos el tamaño de bloque al del warp, podemos encontrarnos con warps incompletos o sin ningún cómputo, desperdiciando capacidad de cálculo.

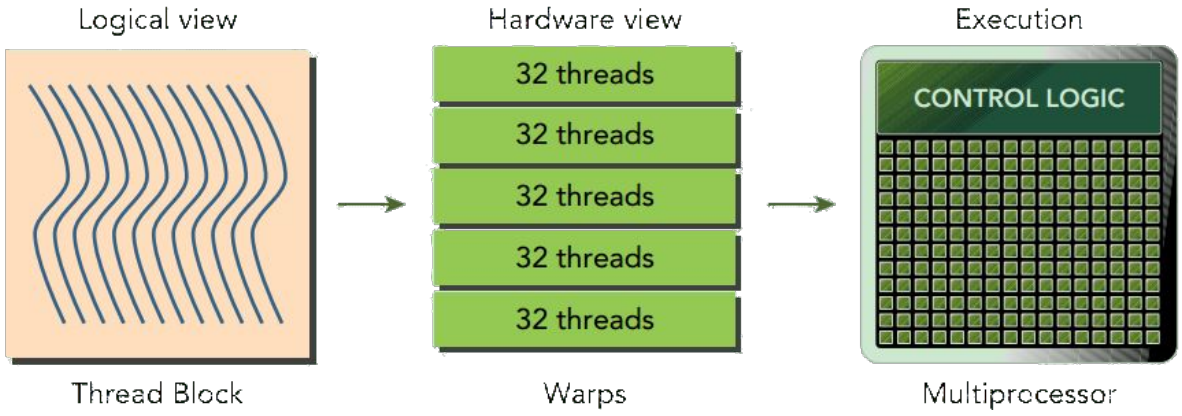

Figura 3.3: Procesamiento paralelo según la perspectiva

Elegir el tamaño correcto entre los múltiplos de 32 dependera de las característias del programa. Por carácter general, un bloque de 32 x 32 es el tamaño más óptimo en la programación paralela.

3.2.3. Intercambio de punteros

Una optimización muy útil en la programación en lenguaje C, es el intercambio de puntero de matrices. La programación en paralelo conlleva en la mayoria de casos, el uso de al menos dos matrices de datos, siendo una de ellas para la entrada de datos, y otra para la salida del resultado. Además por carácter general, son programas que realizan múltiples iteraciones, usando como entrada de datos en cada iteración, el resultado de la última iteración. Esta optimización consiste pues, tal y como se indica en la figura 3.4 , en el intercambio de los punteros de las matrices de entrada y salida, por cada iteración del programa, ahorrando asi la copia de datos que se realizaría de la matriz resultado a la matriz de entrada, necesaria para la siguiente iteración.

```
for (inf i=1; i<iteraciones; i++) {
 kernel(mEntrada, mSalida);

 //La siguiente iteracion necesita los valores de salida como par\'ametros de entrada

 auxiliar = mEntrada;
 mEntrada = mSalida;
 mSalida = auxiliar;
}

//Reajustamos los punteros si es necesario
if (iteraciones % 2 == 0){
 auxiliar = mEntrada;
 mEntrada = mSalida;
 mSalida = auxiliar;
}
```

Figura 3.4: Ejemplo de intercambio de punteros

3.2.4. Transferencia de bordes verticales

Discutimos dos posibles técnicas 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 recepción, 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 discontinuos, 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.

Existen múltiples funciónes más de transpaso de vectores verticales, como *cudaMemcpy2DArrayToArray* o *c udaMemcpyFromArray* , pero obtaremos por cudaMemcpy por ser la opción mas genérica. En trabajos posteriores se podría proceder al analisis del resto de funciones, y realizar un modelo más abstracto que aplique una opción u otra según las características de la estructura de datos, facilitando notablemente la optimización de la transferencia de memorias al programador.

3.2.5. Memoria *pinned*

La transferencia 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 dirección de memoria virtual. 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 reción reservada, sumándole el coste de la transmisión, espera y liberación de memoria de todo este proceso. En cambio, si reservamos memoria *pinned* (memoria no paginada), la comunicación es más rápida debido a que la GPU no necesita hacer la resolución de la dirección 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.

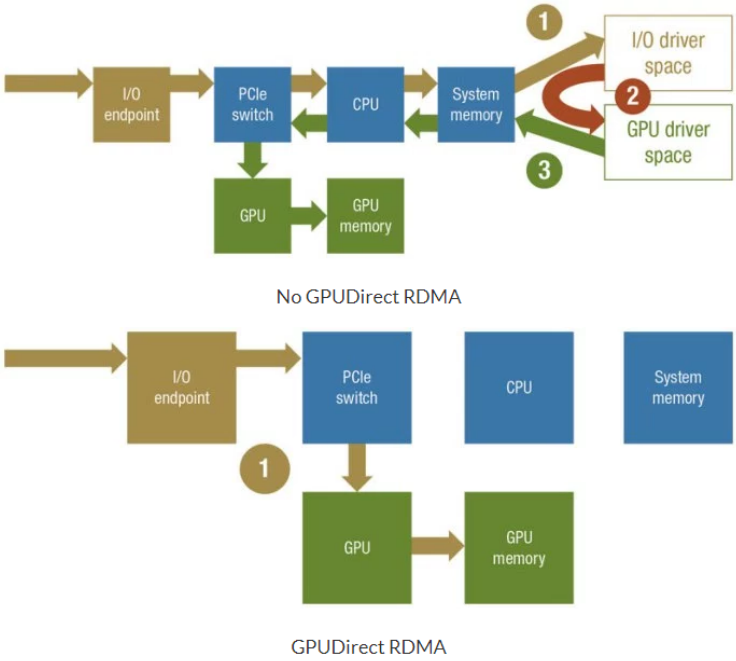

Figura 3.5: Procesamiendo de memoria *pinned*

3.2.6. 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 3.6 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 estaránormalmente 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 transmitirse a través de la conexión entre la CPU0 y la CPU1 para dirigirse a las GPUs que se encuentran en el otro nodo NUMA.

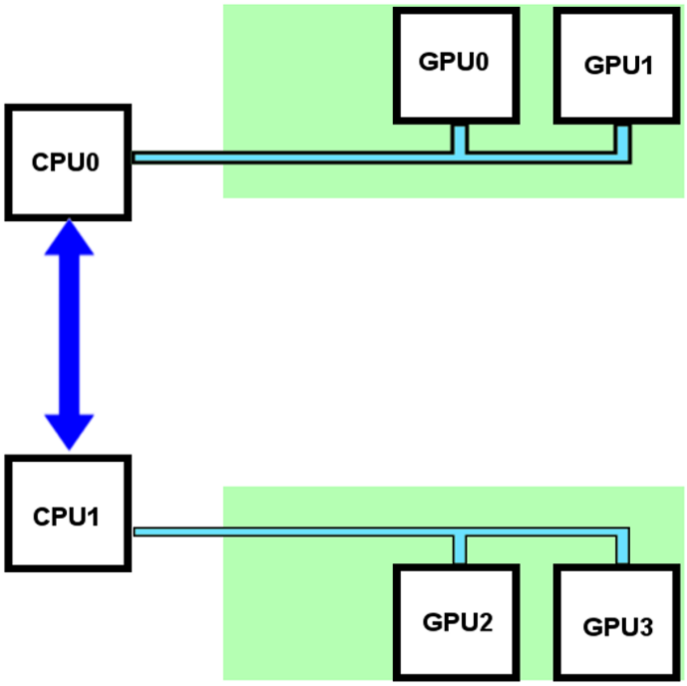

Figura 3.6: Afinidad entre CPU y GPU

3.3. Descripción técnica de los casos de estudio

En este apartado mostraremos un ejemplo detallado de cada caso de estudio, con algunas de las optimizaciones mencionadas en la sección anterior.

3.3.1. Multiplicación de matrices

Tomaremos como caso de estudio técnico, un ejemplo de la multiplicación de 2 matrices distintas en cada iteracion, sin dependencia entre iteraciones. Esto implica que para nuestro problema, dispondremos de dos matrices de entrada, cuyo productor resultante se almacenara en una tercera matriz.

Durante la experimentación se realizaran distintas versiones del programa, diferenciandose por las optimizaciones aplicadas. En este apartado, describiremos el caso más completo, con todas las optimizaciones.

El primer paso es la programación de un kernel que se encargara de procesar el producto de las matrices, tal y como viene en la figura 3.8. El programa funciona de la siquiente manera:

- El primer paso sera la asignación de la afinidad del programa, eligiendo algunas de las CPUs que posean mayor afinidad con la GPU.
- Reservamos la memoria de host y device necesarias, en este caso, tres matrices en el host y seis en el device, ademas de los streams que necesitaremos para realizar las comunicaciones asíncronas. Se reservan seis matrices en el device para poder realizar la copia de forma asíncrona, usando tres de para la entrada de datos y tres para la salida.
- Preparamos un bucle con el número de iteraciones indicado por el programador. Dentro de este, lo primero que se realiza en cada iteración es la copia de los datos de las matrices del host al device, definiendo un stream distinto al que se va a utilizar en los kernels de cómputo.
- Se realiza una sincronización para asegurar el fin de transmision de datos, los cuales necesitaremos para realizar el kernel.

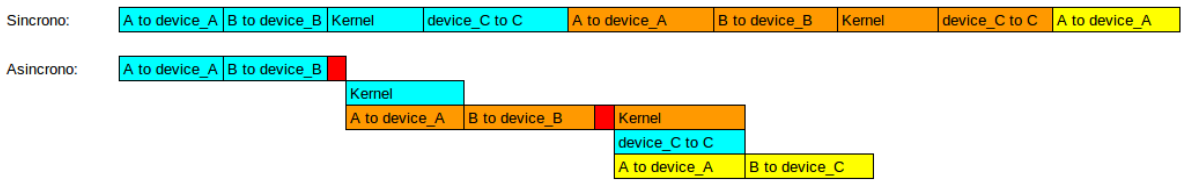

Figura 3.7: Solapamiento de cómputo y comunicación en la multiplicación de matrices

- En este momento, se lanzaran en un stream el computo de la iteración actual, y en un segundo stream, las transmisiones de memoria. Estos son parametros de entrada de la siguiente iteracioń del host al device, y la copia del resultado de la iteracion anterior del device al host (figura 3.7. Todas las transmisiones de datos se realizaran sobre las matrices del device que no estemos utilizando en el kernel actual, ya que podría producir reescritura de datos. El kernel de cómputo que hemos diseñado realiza las siguientes operaciones:

  1. Cada bloque de hilos se encarga del computo de su submatriz. Ya se accede múltiples veces a los datos de las submatrices de A y B, lo primero es la reserva de memoria compartida, para almacenar las submatrices As y Bs. Esto nos permitira un acceso más rápido a los valores de entrada.
  2. De forma paralela, cada hilo del bloque carga 1 elemento de cada submatriz en la memoria compartida.
  3. Una vez sincronizamos el bloque para asegurar la carga de la matriz completa en la memoria compartida, procedemos al cálculo del resultado. Cada hilo realizada el cómputo necesario para el calcular el resultado de su celda.
  4. Una vez finalizada todas las interaciones, se copia los datos en la memoria de device, para que pueda ser copiada hacia el host. Al igual que en los caso anteriores, cada hilo se encargara de la celda correspondiente.

- Una vez finalizado el cómputo del bucle principal, transferimos los datos de la matriz del device al host.

3.3.2. Jacobi 2D

El método Jacobi aplicado a una matriz de dos dimensiones (ver Fig. 3.2) 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 múltiples 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. 3.9). 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 imágen basada en el método de Jacobi, cada bloque de hilos realiza la copia local de la submatriz correspondiente en memoria compartida, además del halo de dicha submatriz [2]. 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.

Para la implementación en un clúster multi-GPU distribuido, será 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 qué evaluamos en este trabajo es el siguiente:

1. Se programan cinco kernels para computar la submatriz del proceso asignada a un dispositivo. Un kernel se encargará de la parte central, sin incluir los bordes. El resto de kernels computarán 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 envío 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 imágenes 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 iteraciones, se se comprueba el número de iteraciones realizadas, para conocer donde se encuentra la matriz resultado 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 operaciones 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.

```
template <int BLOCK_SIZE> __global__ void
matrixMulCUDA(float *C, float *A, float *B, int wA, int wB)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd   = aBegin + wA - 1;
    int aStep  = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep  = BLOCK_SIZE * wB;

    float Csub = 0;

    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep)
    {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Carga de datos en la memoria compartida.
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        __syncthreads();.

        for (int k = 0; k < BLOCK_SIZE; ++k){
            Csub += As[ty][k] * Bs[k][tx];
        }

        __syncthreads();
    }
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Figura 3.8: Kernel CUDA de la multiplicación de matrices

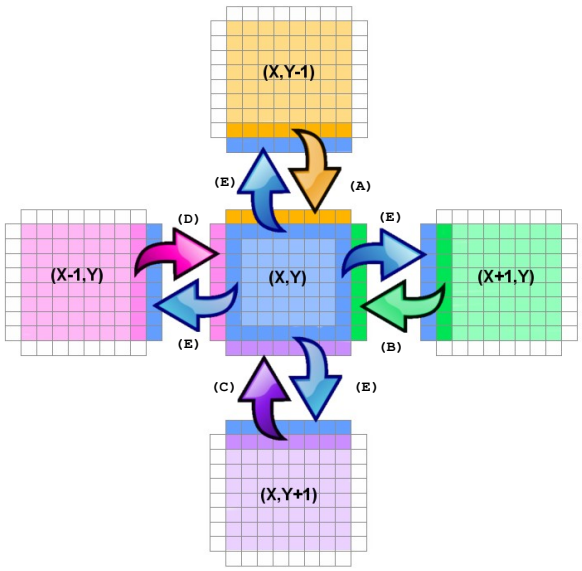

Figura 3.9: Comunicación entre procesos en Jacobi 2D. Se muestran los halos o extensiones de la submatriz asignada al proceso, y los movimientos de datos entre bordes y halos.

```
__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];
}
```

Figura 3.10: Extracto del coígo de Jacobi 2D implementado en un ejemplo de convolución de imágen suministrado con el Toolkit de CUDA

3.4. Implementación de controladores

En esta sección describiremos que pasos hemos tenido que realizar para implementar las técnicas estudiadas en el modelo de controladores. Hemos diseñado una serie de funciones manualmente dentro de la librería de Controladores para óptimizar su uso con nuestro modelo y las optimizaciones estudiadas anteriormente, entre las que encontramos:

- Funciónes de transpaso de datos entre el device y el host, de forma que se realizen sin la función *Deatach*, que es el método por defecto de los Controladores.
- Sistema de streams (cola) para aplicar algoritmos de organización al la cola principal de kernels del Controlador.
- Función de copia parcial de HitTiles, para reducir el transpaso de memoria.

3.4.1. Jacobi2D + Controladores

Hemos utilizado el modelo para la implementación de la abstración mediante controladores. Las características de este modelo nos permiten realizar una prueba mas completa, en comparación a otros modelos como el de la Multiplicación de matrices. Además la poca complejidad de proceso en el modelo teórico del programa, nos permite una rapida comprensión de la computación, centrando nuestra atención a la abstración obtenida y su eficacia.

La implementación de nuestro modelo ha sido realizada de la siguiente manera:

1. En el primer paso definimos las características estructurales del HitTile que vamos a utilizar, mediante el uso de HitShape y HitTopology. En este caso vamos a utilizar un HitShape para definir el tamaño de la submatriz correspondiente al proceso, y un HitTopology de tipo Topology2D correspondiente a la matriz de 2 dimensiones.

2. Realizamos la reserva de memoria de la matriz inicial y una copia, mediante HitTile, que es la estructura de datos sobre la que trabaja los controladores, y las inicializamos con los valores de los bordes.

3. Inicializamos un *HitPattern* para definir el proceso de comunicación de datos que necesitaremos para el paso de las boundaries. Este *HitPattern* nos permite definir un proceso de comunicacion, formado por un conjunto de patrones. En este caso definiremos el envio y recibo de las boundaries mediante cuatro *hit_comSendRecvSelectTag()*, una por cada coordenada, pasando como parametros los HitTile y el HitShape correspondientes.

4. Creamos el controlador, le asignamos una GPU y realizamos el attach de los HitTiles que definimos anteriormente. Al realizar el attach, el controlador reserva y copia los datos de los HitTile a unas variables análogas en la memoria device llamadas *Hit_kTile*.

5. Realizamos el bucle del cómputo principal:

   - Copiamos los valores del HitTile del host al device con los valores actualizados de la última iteración.
   - Lanzamos un kernel que realiza la copia de los datos del HitTile original a la copia en la memoria de device.
   - Lanzamos el kernel de cómputo principal (Fig. 3.11) de forma síncrona.

- Una vez finalizado el kernel, se realiza la transferencia de datos del device al host del HitTile, mediante una llamada que hemos definido de forma manual en la libreria de Controladores, *CAL_CommCopyBoundaryToHost()*, ya que la libreria original no dispone transferencia de datos entre device y host sin realizar una operación de Detach. Esta función que hemos definido realiza una operación de cudaMemcpy del device al host del HitTile que pasamos como parámetro.
- Lanzamos el patrón de comunicación que definimos anteriormente, mediante la llamada de Hitmap *hit_patternDo()*, que realizara el proceso de transmisión de datos.

6. Una vez finalizado todas las iteraciones, procedemos a la transferencia de memoria de device al host.

7. Liberamos la reserva de datos, tanto de las estructuras de Hitmap que hemos definido como las del controlador.

```
CAL_KERNEL_GPU_CHAR_STATIC(Update, 2, medium, medium, medium);
CAL_KERNEL_GPU(Update, 2, OUT, HitTile_float*, dst, IN, HitTile_float*, src){
        int row = threadId.y + 1;
        int col = threadId.x + 1;
        hit_ktileElemAt(dst, 2, row, col) = (
                        hit_ktileElemAt(src, 2, row - 1, col    ) +
                        hit_ktileElemAt(src, 2, row + 1, col    ) +
                        hit_ktileElemAt(src, 2, row    , col - 1) +
                        hit_ktileElemAt(src, 2, row    , col + 1) ) / 4;
}
```

Figura 3.11: Kernel de la libreria de Controladores para el cómputo del método Jacobi

Capítulo 4

Estudio experimental

En esta sección presentamos los resultados de un estudio experimental realizado para comprobar de forma práctica el impacto relativo de las diferentes opciones y optimizaciones consideradas, y para verificar el nivel de escalabilidad conseguido por la solución. Describiremos brevemente en cada caso, la experimentación que se va a realizar, como se ha implementado dentro de su caso de estudio, y un breve análisis de cada experimento de forma aislada.

4.1. Multiplicación de matrices

Hemos desarrollado una implementación propia de la multiplicación de matrices para la realización de esta parte experimental. Partiendo de una versión secuencial del programa, se ha ido paralelizando y optimizando segun las necesidades descritas a continuación. Este caso de estudio nos permitira empezar a comprender el funcionamiento de una programación paralela, el impacto del cómputo y la comunicación, la importancia de mejorar su rendimiento y probar algunas de las optimizaciones teoricas que describimos en el capítulo 3.

4.1.1. Plataforma

Los experimentos de este caso de estudio se han realizado en una máquina local con un solo nodo, que dispone de una tarjeta gráfica de Nvidia para el computo en paralelo. Esta tarjeta gráfica tiene las siguientes características:

- **Arquitectura de GPU:** Pascal (5.2)
- **Memoria de vídeo:** 12 GB G5X
- **Frecuencia de la memoria:** 10 Gbps
- **Frecuencia acelerada real:** 1531 MHz

4.1.2. Comunicación síncrona y asíncrona

La primera experimentación que se prueba es la ejecución paralela de cómputo y comunicación. Esta optimización en la multiplicación de matrices es muy óptima en pequeña escala, perdiendo rapidamente eficiencia a medida que va aumentado el tamaño de carga. Realizaremos 4 pruebas distintas, variando el tamaño de las matrices, comparando el tiempo que tarda en finalizar la versión asíncrona y la versión síncrona. Además, desglorasemos el tiempo de la ejecución síncrona en el tiempo dedicado de cómputo y el tiempo dedicado de comunicación. Respecto al número de iteraciones, en todos los casos las mantendremos

en 500 iteraciones, que nos proporcionarán unos valores más estables respecto a las variaciones que se pueden ocasionar entre las iteraciones del programa.

Tabla 4.1: Resultados al solapar comunicación y computación para matrices de 1000x1000 y 500 iteraciones

	Media	Mínimo	Máximo	Desviación estándar
tCómputo	1,083923697	1,0711841	1,08697589	0,004582947388
tComunicación	1,215737494	0,80644842	1,97699996	0,5102440465
tTotalSincrono	2,299661191	1,89299495	3,04818406	0,5081402841
tTotalAsíncrono	1,485780497	1,07730204	2,53711664	0,5346658423

Con los resultados de la primera prueba (Tabla 4.1), mostrados en la tabla 4.1, podemos observar una gran optimización del rendimiento que nos ofrece esta técnica, reduciendo el tiempo del programa en un 64 % . Esta optimización nunca reducira el tiempo más de un 50 % ya que como se ha explicado, la técnica consiste en realizar la parte computacional y la de comuicación de datos de forma simultanea. Esto implica que el caso ideal es cuando encontramos un coste computacional y de comunicación identicos, reduciendo el coste aproximadamente a la mitad. Además, se produce un coste de tiempo al realizar esta técnica, ya sea preparando las llamadas a las funciones asíncronas equivalentes o la sincronización de datos, por lo que no es posible obtener exactamente el 50 % en la redución de tiempo, si no que el objetivo es obtener el valor más cercano posible.

Por otra parte, vemos que tanto en la versión síncrona y asíncrona, los resultados en los valores máximos y mínimos de la comunicación estan muy alejados de la media, con una desviación estándar similar en ambas. Este valor de la desviación estándar, nos indica que las razones de que los valores no sean estables no han sido producidas por la comunicación asíncrona, si no por otras causas que analizaremos más adelante.

Tabla 4.2: Resultados al solapar comunicación y computación para matrices de 1500x1500 y 500 iteraciones

	Media	Mínimo	Máximo	Desviación estándar
tCómputo	3,392667705	3,37500246	3,41149312	0,01138641295
tComunicación	3,789854593	2,06198032	5,14855397	1,092359679
tTotalSincrono	7,053951345	5,44613521	8,54687194	1,216718847
tTotalAsyncrono	3,838820432	3,39964863	4,58320332	0,5557951848

En la segunda prueba (Tabla 4.3), con una matriz más grande que la anterior, obtenemos unos resultados más óptimos en cuanto al rendimiento obtenido, siendo la reducción del coste de un 54 %. Este mayor rendimiendo es obtenido debido al mayor equilibro que hay entro el peso del cómputo y el peso de la comunicación de datos, teniendo un pequeño de botella en el segundo. Observamos además la desviación estandar de los valores de comunicación se ha incrementado en un orden similar a los resultados, por lo que deducimos que estas variaciones son afectadas por la cantidad de datos que se comunican.

Ha medida que seguimos incrementando el tamaño de la matriz (Tabla 4.2), vemos que el peso del cómputo se incrementa en un orden mayor que el de las comunicaciones, siendo en esta prueba donde el cuello de botella que se encontraba en las comunicaciones, pasa a estar ahora en el peso del cómputo. En este caso, la optimización sigue aproximadamente sobre el 54 %.

En esta última prueba (Tabla 4.4), observamos claramente el incremento de mayor orden del peso del computo, siendo este casi el doble del peso de la comunicación. Por tanto, a medida que vayamos incrementando el tamaño de la matriz, nos encontraremos con una mayor diferencia entre el cómputo y la

Tabla 4.3: Resultados al solapar comunicación y computación para matrices de 2000x2000 y 500 iteraciones

	Media	Mínimo	Máximo	Desviación estándar
tCómputo	8,265151934	8,20382301	8,30069843	0,03985782526
tComunicación	7,266345555	3,9046716	9,71185624	2,353567235
tTotalSincrono	15,53149749	12,10849461	17,99203074	2,360810333
tTotalAsyncrono	8,419728988	8,1618545	9,66756341	0,4625430169

Tabla 4.4: Resultados al solapar comunicación y computación para matrices de 2500x2500 y 500 iteraciones

	Media	Mínimo	Máximo	Desviación estándar
tCómputo	16,56783691	16,43323562	16,79730395	0,1272761114
tComunicación	9,446852215	5,94960866	14,52472202	3,658129731
tTotalSincrono	26,01468912	22,55347614	31,03799521	3,577108246
tTotalAsyncrono	17,2088918	16,09049844	18,5908821	0,945318933

comunicación, lo que nos producira un menor rendimiento de la optimización. Además, observamos que la desviación de los valores producidos por la comunicación se han camuflado en el tiempo de cómputo.

Concluimos en esta sección de la experimentación, que esta técnica de solapar comunicación y computación es muy eficaz para matrices de un orden de 1500 x 1500, reduciendo su eficacia a medida que incrementamos el tamaño. Este orden variara ligeramente según las características de las plataforma donde se realice la experimentación. Aunque es una optimización muy eficaz en matrices pequeñas, en el uso de la aplicación en casos reales, donde se realizan multiplicaciónes de matrices del orden de 100.000 x 100.000, obtendremos una reducción del tiempo cerca del 0 %. Aun asi, esta experimentación nos ayudara a comprender y aplicar de forma más eficaz esta técnica, en otros casos de estudio donde podamos obtener una mejora en el rendimiento más escalada.

4.1.3. Memoria *Pinned*

Los altos valores en la desviación estándar de la última prueba nos incrementan en gran medida los tiempos del programa. Esto nos lleva a averiguar que esta produciendo estas variaciones. Estas variaciones estaban producidas por el tiempo de comunicación del programa, por lo que buscamos técnicas que afectene en la comunicación de datos. La primera técnica que realizamos para intentar solucionar este problema es el uso de memoria *pinned*. Se espera que esta técnica la mejore de la velocidad en la comunación de los datos, reduciendo a su vez las desviaciones producidas por esta.

Tabla 4.5: Resultados al aplicar memoria emphpinned a la multiplicación de matrices

Tam. matriz	tiempo mem. no *pinned*	tiempo mem. *pinned*	Desviación est. no *pinned*	Des. est. *pinned*
1000x1000	1,485780497	1,171199215	0,5346658423	0,2385718362
1500x1500	3,838820432	3,514760918	0,5557951848	0,2254708093
2000x2000	8,419728988	8,246096816	0,4625430169	0,06284979532
2500x2500	17,2088918	16,78468652	0,945318933	0,5582991088

Lo resultados que obtenemos (Tabla 4.5) corroran la mejora que ofrece la aplicación práctica de esta técnica. Aunque en los tiempos totales obtenemos unos valores ligeramente menos, hemos logrado reducir la desviación estandar de los resultados a la mitad. Para la obtención de estos datos, se realizaron muestreos de 10 elementos. Aun así, vemos que dentro de los resultados obtenidos, los valores en la

reducción de la desviación estándar en la matriz de 2000 x 2000 no siguen el mismo patrón que el resto de casos, reduciendose la desviación estandar hasta un 90 %. Esto nos lleva a pensar que las causas de esta alta desviación de los datos no ha sido solucionada por esta optimicación, aunque si se han mitigado.

4.1.4. Afinidad entre CPU y GPU

Siguiendo la misma idea de reducir el coste de las comunicaciones, procedemos a probar la afinidad entre la CPU y la GPU. El procedimiento para definir la afinidad de nuestro programa se realiza mediante la llamada a *sched_setaffinity*, que encontramos dentro de la libreria *sched.h* de C. El número de CPU que debemos asignar dependera de la plataforma donde se ejecute la aplicación. Para nuestra experimentación, realizaremos las pruebas con una afinidad sin definir, otra asignando una buena afinidad y por último asignando una mala afinidad.

Tabla 4.6: Resultados al definir afinidad a la multiplicación de matrices

	Tiempo comunicación	Desviación estándar
Afinidad sin definir	0,632966471	0,4102698328
Buena afinidad	0,490997681	0,0007073336749
Mala afinidad	1,765650381	0,07089137166

Tal y como vemos en los resultados, con esta optimización hemos estabilizado los tiempo de comunización, los cuales variaban según la afinidad que tuviesen, llegando a ser 100 veces mayor el tiempo de comunicación con una mala afinidad respecto a los valores de tiempo de una buena afinidad. En el caso de que no definamos un afinidad al programa, se le asignara a este una CPU de forma aleatoria en tiempo de ejecución, por lo que los valores que obtendremos seran mayores o no según la afinidad obtenida. Esta aleatoriedad se comprueba con los datos que obtenemos al asignar una afinidad, sea buena o mala, conseguiendo reducir la desviación estandar del tiempo en ambos casos.

4.2. Jacobi 2D

Hemos escogido una implementación de Jacobi2D, con una partición de datos clásica y sencilla: bloques bidimensionales homogéneos. 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. Hemos escogido como kernel de base una traducción directa de la función de actualización de un elemento como se muestra en la figura 4.1.

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 3.1.2, probando las diferentes opciones y optimizaciones detalladas en la sección 3.2.

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 *double*, ya que es el tipo más adecuado para este tipo de aplicaciones científicas. En la experimentación utilizamos tamaños de problema que derivan en

```
__global__ void jacobiCUDA
(double *C, double *A, int x, int y){

#define elem(mat, idx1, idx2, size)  \
(mat[(idx1)*size+(idx2)])

int tx = threadIdx.x;
int ty = threadIdx.y;
int bx = blockIdx.x;
int by = blockIdx.y;
int bdx = blockDim.x;
int bdy = blockDim.y;

int idx = tx + bx * bdx;
int idy = ty + by * bdy;

if((idx>1)&&(idy>1)&&(idy<y-2)&&(idx<x-2)){
 elem(C,idx,idy,y) =
 ( elem( A,idx-1,idy,y )  +
    elem( A,idx+1,idy,y ) +
    elem( A,idx,idy-1,y ) +
    elem( A,idx,idy+1,y ) ) / 4;
}
}
```

Figura 4.1: Código kernel de actualización Jacobi en 2D

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.

4.2.1. 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 heterogéneos con nodos de diferentes capacidades. El la tabla 4.7 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, 32 GB de RAM, y dos GPUs TESLA C2070, con un rendimiento en coma flotante de doble precisión (pico) de 515 GFlops, memoria dedicada de 6GB GDDR5 y una velocidad de 1.5 GHz. La tecnología de red de interconexión es Infiniband QDR (40Gb/s) y FDR (56Gb/s).

4.2.2. Comunicación síncrona y asíncrona

La comunicación asíncrona de datos entre host y dispositivos, solapandose con la computación en cada iteración, es una de las optimizaciones com más impacto en cómputos iterativos, como es en el caso de Jacobi 2D. Esto es factible en el caso de nuestra computación stencil, que no presenta dependencias entre

Tabla 4.7: Clúster Trasgo: Máquinas y características

	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

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.

Tabla 4.8: Tabla con los tiempos de computación y comunicación, con 50 iteraciones (en segundos)

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

Tabla 4.9: Tabla con modelo síncrono y aśincrono, con 50 iteraciones en hydra (en segundos)

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

En la tabla 4.8 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 *Hydra*, utilizand las 4 GPUs disponibles. Observamos que los tiempos de cómputo y comunicación son muy similares, creciendo ambos proporcionalmente con el tamaño de problema.

En la tabla 4.9 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 cómputo 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, observándose tiempos totales que tienen apenas un incremento de entre un 1 % y un 2 % sobre los tiempos de comunicación de la versión de referencia.

4.2.3. 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 4.10 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 optimizaciones 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 4.10: 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

4.2.4. 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 aplicación cuando se amplia el número 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 cómputo 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,

Para obtener los tiempos más fiables posibles en las pruebas de escalabilidad, utilizamos un tamaño de matriz grande para tener una carga notable y un número de repeticiones suficientes para regular el número de comunicaciones entre procesos. El objetivo es estabilizar el efecto que tienen las comunicaciones y la computación en el rendimiento del programa, y aislar de esta manera las variaciones de tiempo producidas por el incremento de los nodos y procesos.

En el caso del clúster Trasgo, empezamos usando cores de una única máquina, incrementando el número de cores de cuatro en cuatro, hasta completar su capacidad. Se continuan introduciendo cores de cuatro en cuatro de las siguientes máquinas, siempre completando 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 estón 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 homogéneas el orden en que se añaden los nodos no es relevante.

Tanto en la figura 4.2, como en la figura 4.3, 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 situación de alta afinidad, donde los cores que se están 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 ligerésimo incremento hacia el final en el caso del clúster Trasgo al introducirse

tencnología de red más limitada (Ethernet 1Mb).

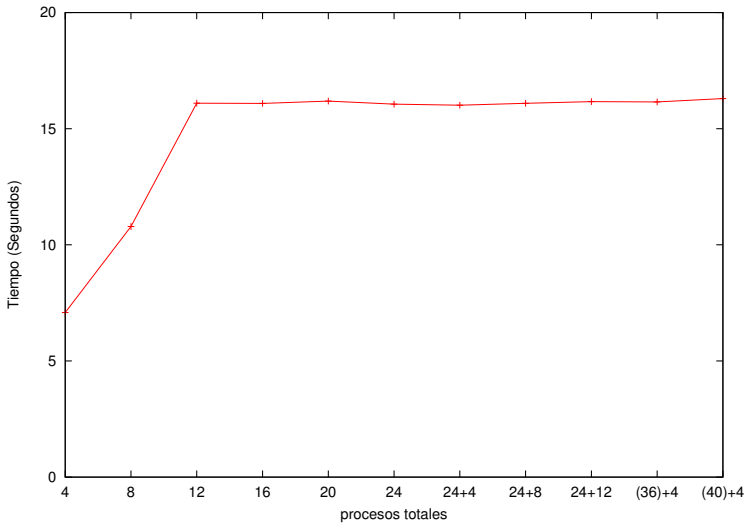

Figura 4.2: clúster Trasgo: Weak scaling con número de procesadores

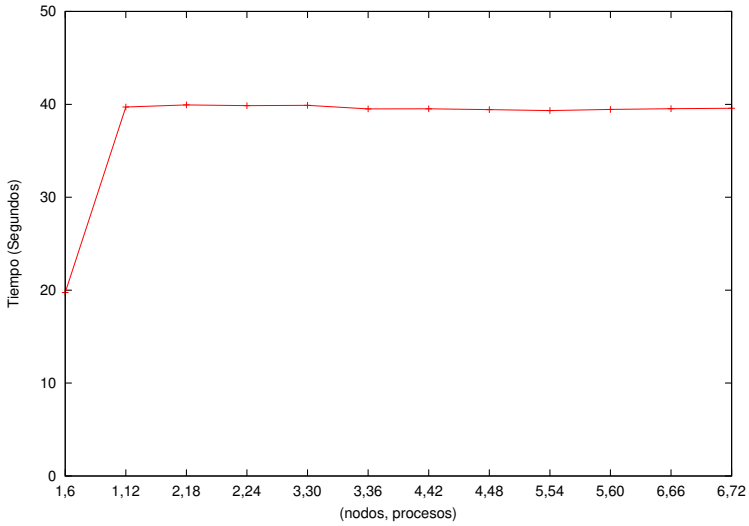

Figura 4.3: clúster CETA: Weak scaling con número de procesadores

En la segunda parte del estudio comprobamos la escalabilidad para un tamaño de problema fijo (*strong scaling*). En la figura 4.4 se muestran los resultados 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 basada en bus, como es Ethernet, la cantidad y frecuencia de las comunicaciones de este tipo de aplicaciones no llega a ser suficiente Las irregularidades 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.

4.2.5. 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, reduciéndose los tiempos de cómputo e incrementándose los de comunicación debido a la introducción de las transferencias entre la jerarquía, de memoria del dispositivo y el host.

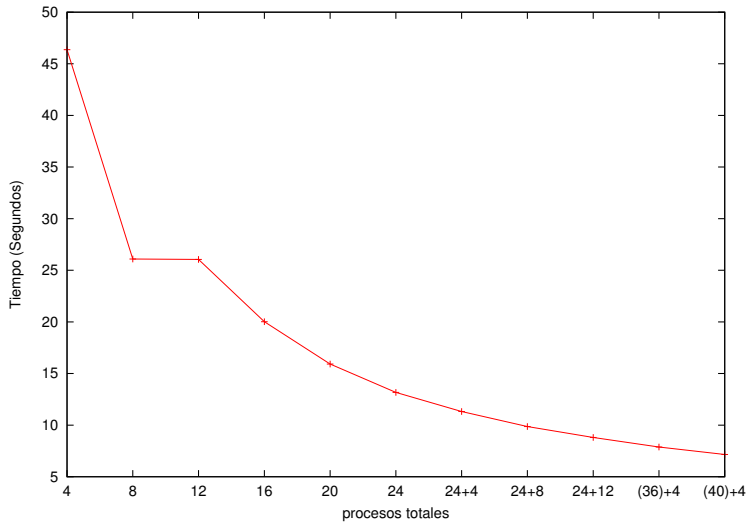

Figura 4.4: clúster Trasgo: Strong scaling con número de procesadores

En la figura 4.5 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).

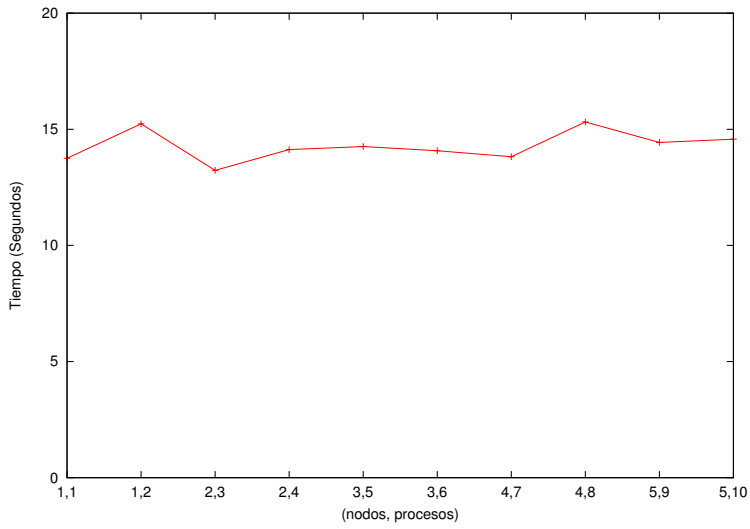

Figura 4.5: clúster CETA: Weak scaling con GPUs en nodos distribuidos

En la figura 4.6 mostramos los resultados de escalabilidad fuerte en el clúster 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).

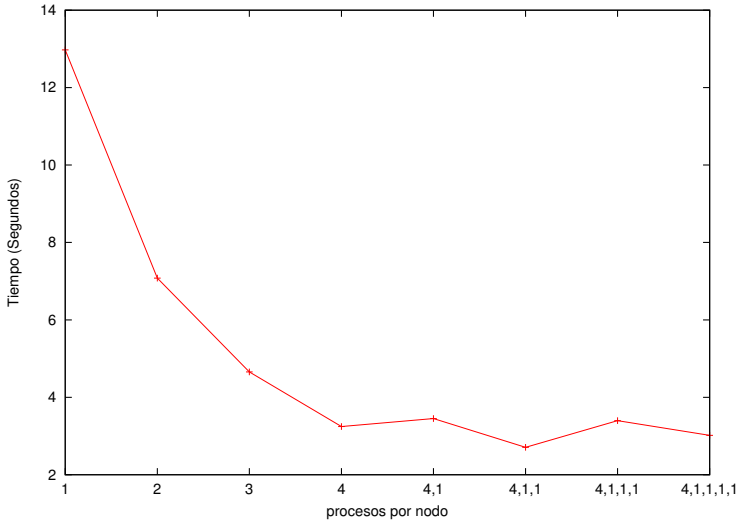

Figura 4.6: clúster Trasgo: Strong scaling con GPUs en nodos distribuidos

4.3. Implementación de controladores y Hitmap

Para el desarrollo del estudio experimental de los controladores, vamos a analizar los valores de complejidad ciclómatica, uno de los métricas mas aceptadas para el análisis de la complejidad lógica de un programa.

Los resultados de este experimento no varian segun la plataforma sobre la que se realice, ya que es una experimentación lógica del programa.

4.3.1. Resultados

Para obtener una comparativa de la reducción de la complejidad ciclomática, realizamos las pruebas sobre tres versiónes del método de jacobi que se ha desarrollado durante este trabajo: la versión original implementanda solo en MPI, la segunda versión agregando el uso de CUDA para el cómputo, y la última versión desarrollada con los Controladores. Podemos ver los resultados desarrollados del análisis en las tablas 4.11, 4.12 y 4.13. Podemos observar en estas tablas los resultados desagregados de cada programa por función.

La tabla muestra los siguientes métricas:

- **NLOC:** número de lineas de código.
- **CCN:** grado de complejidad ciclomática.
- **token:** número de variable.
- **PARAM:** número de parametros.
- **location:** nombre de la función.

Tabla 4.11: Análisis de la complejidad ciclomática de jacobi2D con MPI

NLOC	CCN	token	PARAM	location
24	5	142	1	factors2D@64-95
245	72	2994	2	main@101-454

Tabla 4.12: Análisis de la complejidad ciclomática de jacobi2D con CUDA+MPI

NLOC	CCN	token	PARAM	location
14	3	105	3	initMatrixCUDA@66-86
14	5	177	4	jacobiCUDA@156-177
334	70	3712	2	main@218-667

Tabla 4.13: Análisis de la complejidad ciclomática de jacobi2D con Controladores

NLOC	CCN	token	PARAM	location
10	3	73	3	hit_shapeExpand@57-70
7	3	65	4	hit_shapeDimExpand@73-84
5	1	58	8	CAL_KERNEL_GPU@93-97
5	1	56	8	CAL_KERNEL_GPU@103-107
9	1	107	8	CAL_KERNEL_GPU@111-121
18	5	210	1	initMatrix@138-190
158	14	1442	2	main@193-459

Con los resultados de cada caso, realizamos una síntesis conjunta y realizamos un análisis de los datos que obtenemos (tabla 4.14.

Como podemos observar, vemos que la versión de CUDA+MPI incrementa la complejidad su programación en grán medida respecto a su versión de MPI, en orden de obtener un mayor rendimiento en tiempo. Esto nos hace plantear si realmente es rentable realizar un modelo de MPI+CUDA.

Gracias a nuestro modelo abstracto con controladores, logramos reducir esta complejidad de forma drástica, llegando a la complejidad ciclomática a la mitad. Esto es debido a sus capacidad de ocultar la complejidad de CUDA mediante el uso de Controladores, y la complejidad de MPI con el uso de estructuras Hitmap.

Tabla 4.14: Comparación de la complejidad ciclomática de jacobi2D

	Total nloc	CCN	token	Fun Cnt
MPI	276	77	3136	2
MPI+CUDA	393	83	4136	4
Controlador	228	33	2993	7

Capítulo 5

Conclusiones y Trabajo Futuro

En este trabajo se ha realizado el estudio e implementación de un modelo de optimización y alta abstración de cómputo paralelo mediante el uso de CUDA y Controladores. Partiendo de los objetivos iniciales que habiamos propuesto al inicio de esta trabajo, hemos conseguido cumplir satisfactoriamente todos ellos: (1) Hemos logrado reducir el coste de comunicación de los casos de estudio que hemos elegido mediante técnicas de solapamiento y uso óptimo de las memorias disponibles en las GPUs; (2) Redución de la complejidad de programción de un 50 % mediante el uso de estructuras abstractas; (3) Expandir mi conocimiento inicial de la computación paralela, en gran medida gracias al estudio intensivo que se ha realizado del estado del arte, permitiendome conocer tecnologías que no habia visto nunca. Con todo esto, mi interés por esta tecnología a incrementado de forma exponencial, tanto que probablemente seguire su estudio en el futuro durante la docencia del master y el posterior doctorado.

Aun con los objetivos de este trabajo cumplidos, este modelo aun tiene mucho potencial para ser mejorado, dando opción a muchas vias de desarrollo partiendo este trabajo como base, como por ejemplo aplicar otros modelos de abstración o aplicar otras API de desarrollo de programación paralela, y realizar la comparativa de con este modelo. Otros de trabajos interesantes a realizar seria un *benchmark* de este modelo con una aplicación real de cómputo paralelo y su comparación con otros modelos. Además el desarrollo de la computación paralela seguira incrementandose, permitiendonos creando cada vez nuevas técnicas y probar nuevas plataformas de desarrollo, para mejorar nuestro modelo de forma periódica.

El desarrollo de esta práctica ha sido publicado como un artículo para las *XXVIII Jornadas de Paralelismo (JP2017)*, donde se presentara el 20 de septiembre de 2017.

Referencias

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

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

[3] A. Moreton-Fernandez, A. Gonzalez-Escribano, and D. R. Llanos, "Exploiting distributed and shared memory hierarchies with hitmap," in *2014 International Conference on High Performance Computing Simulation (HPCS)*, July 2014, pp. 278–286.

[4] A. 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*, vol. 0, no. 0, pp. 1094342017702962, 0.

[5] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, New York, NY, USA, 2014, PGAS '14, pp. 6:1–6:11, ACM.

[6] James Dinan, Pavan Balaji, Jeff R. Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju, "Supporting the global arrays pgas model using mpi one-sided communication," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, Washington, DC, USA, 2012, IPDPS '12, pp. 739–750, IEEE Computer Society.

[7] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and Dhabaleswar K. Panda, "Optimizing mpi communication on multi-gpu systems using cuda inter-process communication," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, Washington, DC, USA, 2012, IPDPSW '12, pp. 1848–1857, IEEE Computer Society.

[8] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer, "Libwater: Heterogeneous distributed computing made easy," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, New York, NY, USA, 2013, ICS '13, pp. 161–172, ACM.

[9] 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.

Capítulo 6

Anexo I

El CD entregado cuenta con los siguientes directorios:

- Memoria: contiene esta memoria en formato PDF.
- Ficheros de prueba: Contiene los distintos fichero fuente desarrollados en el trabajo.
- Data: Ficheros .dat con los resultados de las pruebas realizadas.