Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática
(Mención en Computación)

Solapamiento transparente de tareas de comunicación y computación en dispositivos GPU CUDA

Alumno:
D. Ismael José Taboada Rodero

Tutores:
Dr. Yuri Torres de la Sierra
Dr. Arturo González Escribano

Dedicado a mis padres, mi hermano, familia y amigos, y a todos los que me habéis acompañado en este trayecto de mi vida. En especial, se lo dedico a mi abuelo Manuel al siempre recordaré con mucho cariño.

Resumen

La computación paralela ha adquirido importancia significativa en diversos campos de la ciencia y la tecnología, presente en el incremento de trabajos de investigación donde se consigue mejorar el rendimiento del uso de los recursos de los diferentes dispositivos de un sistema. Estos trabajos son, como ejemplo, el entrenamiento de grandes redes neuronales dedicadas al aprendizaje profundo, las simulaciones de fluidos que exigen cada vez más precisión, o la creciente tendencia de uso de algoritmos paralelos para el minado de cripto-monedas.

Actualmente, la programación paralela basa sus soluciones sobre todos los tipos de hardware existentes. La construcción de granjas de computación –como las presentes en la lista TOP 500–, la fabricación de unidades de proceso cada vez con mayor número de núcleos, el uso de unidades de computo gráfico –también conocidas como GPU– para cálculos genéricos (*General-Purpose Computing on Graphics Processing Units* ó GP-GPU) o el diseño de nuevos co-procesadores como Xeon Phi, muestran el amplio abanico de máquinas y dispositivos donde se realizan las ejecuciones de dichos programas.

En particular, la diversidad de dispositivos GPU (compatibles con la computación de propósito general) hace que se deba implementar soluciones específicas pensando en los dispositivos donde va a ser ejecutadas. La programación de estos dispositivos necesita de unos conocimientos de alto nivel sobre la arquitectura y las librerías de desarrollo de cada dispositivo, las cuales se renuevan cada poco tiempo. Esta dificultad aumenta cuando se pretende explotar, de forma eficiente, los diferentes recursos hardware de un dispositivo.

Este trabajo propone un modelo de programación que permite el solapamiento de operaciones de comunicación y computación en dispositivos GPU mejorando el rendimiento de las aplicaciones. Nuestro estudio experimental muestra que este modelo oculta, de forma automática y transparente para el usuario, el solapamiento de estas operaciones ahorrando esfuerzo de desarrollo al usuario. Con este modelo se obtiene hasta un 50 % de mejora de rendimiento comparado con una implementación puramente secuencial.

Abstract

High performance computing has a significant importance in diverse fields of science and technology, as shown in the increment of the number of researchs which obtain performance optimization in the use of different devices of a system. These researchs are such as the train of huge neuronal networks for deep learning, simulations of fluids which need high accuracy, or the rising tendency use of parallel algorithms for mining of crypto-currency.

Nowadays, high perfomance computing works over all types of existent hardware. The construction of server farms –as shown in TOP500 ranking–, the production of units of processing with large number of cores, the use of graphics processing units –as known as GPU– for generic computing (*General-Purporse Computing on Graphics Processing Units* or GP-GPU) or the desing of new co-processors as Xeon Phi, show the wide variety of machines and devices where parallel programs are executed.

In particular, the diversity of GPU devices (compatible with general purpose computing) make developers to implement specific solutions based on the device it will be used. Programming this particular solutions needs high acknowledge of the architecture and development kit of each device, which are renewed every short term of time. However, all this technologies and libraries demand knowledge of specs of devices on which computation is going to be done. This difficult increase when we want to exploit efficiently hardware resources of a device.

This work propose a programming model which allow automatic overlap of operations of communication and computation in GPU devices improving the performance of parallel applications. Our experimental study show that this model hides, automatically and transparently for the user, the overlap of these operations saving effort of development to the user. With this model we obtain up to 50 % of performance improvement compared to a pure sequential model.

Tabla de Contenidos

Lista de figuras

Lista de tablas

Lista de algoritmos

Capítulo 1

Introducción

Este capítulo introduce los siguientes aspectos:

- El origen de la programación de alto rendimiento y sus bases en la computación paralela.
- El contexto y motivación del trabajo.
- El planteamiento del problema encontrado.
- Los objetivos de la solución propuesta.
- La estructura del documento.

1.1. Introducción a HPC

La programación tradicional orienta la codificación de las aplicaciones hacia la computación, en serie o secuencial, de las diferentes instrucciones de una aplicación. Los primeros computadores, en 1950, están compuestos de un único procesador (*mono-core*) que ejecutaba las instrucciones de un programa. Esta arquitectura no permiten tener múltiples operaciones ejecutándose al mismo tiempo. Esa limitación dio lugar al desarrollo de distintas disciplinas de computación tales como la computación distribuida, la computación concurrente o la computación paralela.

En 1960 tiene lugar el inicio de la disciplina de la computación concurrente [8]. Esta disciplina desarrolla las bases necesarias para que una única unidad central de proceso (CPU) pudiese ejecutar varias operaciones de forma independiente. Estas operaciones concurrentes también comparten otros recursos como los accesos a dispositivos de entrada y salida (teclado, conexión de red, ...), o el acceso a dispositivos de almacenamiento o memoria RAM. La computación concurrente modela la interacciones y comunicaciones entre estas operaciones para su coordinación.

La computación paralela surge en 1958 (anterior a la computación concurrente) sobre la necesidad de ejecutar cálculos sobre unidades de cómputo independientes en más de una única unidad de proceso (*core*) de una CPU, para mejorar el rendimiento de programas. En 1962 se construyeron las primeras unidades de proceso compuestas por varios procesadores (*multi-core*) que accedían a módulos compartidos de memoria. Sobre esta nueva arquitectura la computación paralela estudia la ejecución a la vez de operaciones por nodos independientes de procesamiento. En 1965, se construye la primera super-computadora compuesta por una

unidad de procesamiento (CPU) y 10 unidades periféricas de procesamiento. En 1967, se publica "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities" [4] considerado el libro blanco de la computación paralela. Esta disciplina estudia también la computación paralela en clusters o redes de computadoras, construidas por primera vez en 1969.

La computación de alto rendimiento (*HPC*) surge del diseño y desarrollo de nuevas computadoras que cada vez ofrecen más potencia de cálculo. El concepto HPC se basa en la computación paralela, la computación concurrente y la computación distribuida para mejorar la potencia de cálculo de programas ejecutados en super-computadoras. Esta computación también tiene efectos sobre varios campos de la ciencia y la tecnología actual, al reducir el tiempo de computación de muchos problemas de simulación y experimentación. Problemas de distintas áreas como la predicción meteorológica, el desenrollado del ADN o el entrenamiento de modelos automáticos de inteligencia artificial, se ven beneficiados por la alta capacidad de estas super-computadoras. La lista TOP500 [9] recoge las super-computadoras existentes a nivel mundial.

En 1976 se introduce la interfaz gráfica en las computadoras. La imagen de la interfaz sobre la pantalla necesita cálculos complejos, esto hace necesaria incluir un componente hardware especializado. Este componente son los circuitos especializados en procesamiento de vídeo, que más tarde pasan a ser las unidades de procesamiento gráfico (*GPU*). Estos componentes tienen una evolución con la necesidad de producir cada vez cálculos gráficos más costosos de computar. La simulación de entornos 3D en las pantallas de las computadoras provoca que estos componentes deban tener más potencia de cálculo. Dado que estos componentes poseen gran capacidad de cómputo se empezó a utilizar para otros cálculos no gráficos. A partir del año 2000, estos componentes se pasan a considerar co-procesadores (microprocesadores utilizados como suplemento de las funciones del procesador principal) o aceleradoras de la computadora. A diferencia de las CPU, estas aceleradoras contienen un gran número de unidades de cómputo para explotar el paralelismo inherente (*many-core*) de las aplicaciones. Esta evolución en la arquitectura genera un salto de potencia de cálculo grande entre las CPU y las aceleradoras que va incrementando en el tiempo.

1.2. Contexto y motivación

El uso de aceleradores hardware de alto rendimiento tales como, las unidades de procesamiento gráfico (GPU), ha ido en creciente aumento en los sistemas de super-computación. Esta tendencia en fácilmente apreciable en la lista de computadoras mostradas por la clasificación TOP500 [9]. Estos super-computadores también utilizan otro tipo de aceleradoras tales como los co-procesadores (microprocesador de un ordenador utilizado como suplemento de las funciones de la CPU, p.e. Xeon Phi) o las FPGA (matrices de puertas programables). Esta diversidad de dispositivos de cómputo hace que los entornos de computación se denominen entornos heterogéneos.

Actualmente existe una amplia variedad de modelos y arquitecturas de dispositivos GPU. Este conjunto de dispositivos GPU heterogéneos hace que las soluciones diseñadas se deban implementar pensando en el hardware específico donde van a ser ejecutadas. Por ello, el desarrollador necesita unos conocimientos elevados sobre la arquitectura subyacente del dispositivo. Además, es necesario aplicar conocimientos sobre las librerías y entornos de desarrollo existentes para cada dispositivo, las cuales se encuentra en constante

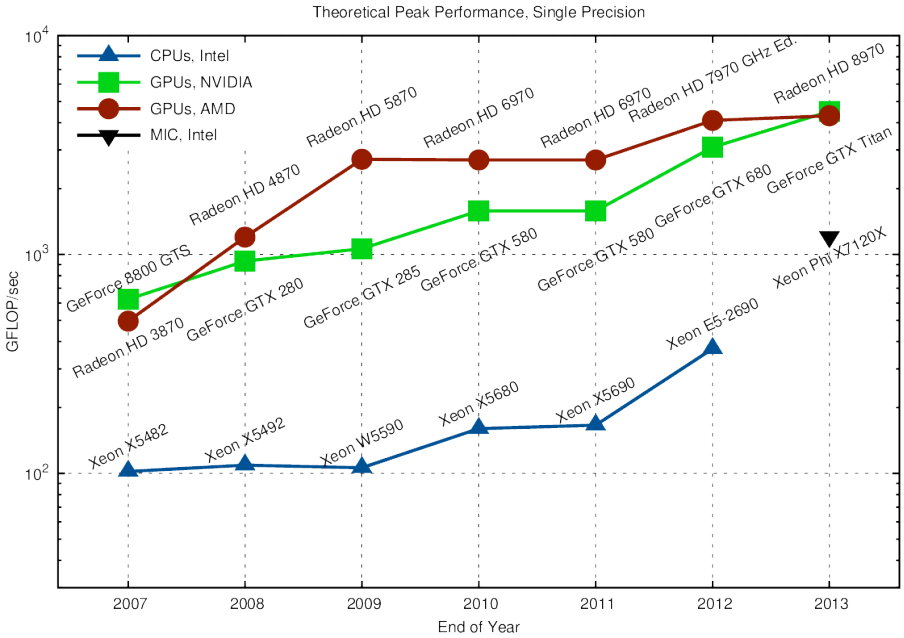


Figura 1.1: Comparación de rendimiento en operaciones de punto flotante entre CPU y aceleradoras hardware.

evolución. Existen librerías y arquitecturas que ayudan al desarrollo de aplicaciones GP-GPU como: el modelo CUDA [15] que ofrece una arquitectura y una capa para programación GP-GPU sobre dispositivos de NVIDIA; OpenCL [23] es una librería que ofrece una interfaz de programación sobre las GPU de diferentes fabricantes y sobre entornos heterogéneos con múltiples dispositivos CPU o aceleradoras; existen otras librerías como Close to Metal (originalmente THIN – Thin Hardware INterface – y posteriormente AMD APP SDK, ofrece un entorno para computación paralela sobre aceleradoras AMD) o BrookGL [10] (librería en C que facilita la programación de aplicaciones paralelas sobre hardware de ATI y NVIDIA). Aunque todas ellas exigen altos conocimientos específicos de computación paralela y se renuevan cada pocos años, forzando al usuario a mantener sus conocimientos actualizados. Además, el uso de técnicas de optimización para mejorar el rendimiento de estas aceleradoras exigen un alto esfuerzo de desarrollo para el usuario y son tendentes a fallos.

1.2.1. Planteamiento del problema

Existen diferentes estrategias para la mejora de rendimiento de aplicaciones paralelas. Entre ellas destacamos las siguientes: a) la adaptación de la programación de la aplicación a la arquitectura sobre la que se va a ejecutar, esta estrategia exige altos conocimientos de la arquitectura y limita la ejecución de la aplicación a máquina objetivo, dificultando la portabilidad de la solución; b) la partición del problema en bloques, distribuyendo los datos sobre diferentes unidades de cómputo, esta estrategia se conoce como *tiling* [25]; c) la creación de procesos y distribución de las iteraciones de un bucle en diferentes unidades de cómputo según una política de gestión [26]; y d) el solapamiento de operaciones de transferencia de datos y computación [7].

El solapamiento de comunicación y computación es una estrategia ya ha sido utilizada en paralelismo

sobre sistemas multi-core o en clusters. Actualmente existe tecnología como CUDA que permiten este tipo solapamiento de operaciones sobre dispositivos GPU NVIDIA. Además, existen sistemas automáticos para las estrategias de partición [12] y reparto de iteraciones de bucles [5, 19, 24] basados en las características del entorno y en la carga de trabajo de la aplicación. Para el solapamiento de operaciones de comunicación y computación únicamente existen modelos que trabajan sobre operaciones básicas de una ejecución [13]. Estos modelos necesitan que el usuario determine los puntos de solapamiento y no analizan, de manera automática, la ejecución completa de la aplicación paralela.

1.2.2. Objetivos

Este trabajo propone un modelo de programación que permite el solapamiento de operaciones de comunicación y computación de forma automática y transparente para el usuario sobre dispositivos GPU. Este modelo mejora el uso de los recursos hardware, de forma transparente para el usuario, asegurando siempre la consistencia secuencial de los datos [1]. Se realiza un prototipo para la programación de aceleradoras NVIDIA. Este prototipo se basa en el análisis de dependencias entre operaciones. Además, se presenta un estudio experimental con dos casos de prueba que muestran cómo el modelo propuesto mejora el rendimiento de las aplicaciones. El modelo propuesto combina las disciplinas de computación paralela y concurrente, aplicando las gestiones de comunicación y sincronización de los procesos de una aplicación paralela.

Los objetivos específicos de este trabajo son los siguientes:

1. Estudiar y comprender los trabajos existentes.
2. Estudiar las librerías Hitmap [12] y Controllers [17] sobre las que se asienta este trabajo.
3. Diseñar e implementar un prototipo del modelo propuesto.
4. Diseñar, implementar y ejecutar casos de experimentación que permitan validar el modelo.
5. Obtener resultados y extraer conclusiones de la experimentación realizada.

1.3. Estructura del documento

El resto del trabajo se organiza en los siguientes capítulos: El capítulo 2 presenta introduce conceptos básicos de computación paralela y algunos trabajos relacionados como las librerías las librerías Hitmap y Controllers. El capítulo 3 muestra las especificaciones del modelo propuesto. El capítulo 4 describe la implementación realizada del modelo sobre la libería Controllers que permite el solapamiento de comunicaciones y computación, de forma transparente, para el usuario. El capítulo 5 expone los resultados experimentales obtenidos sobre los casos de estudios realizados que permitan validar el modelo propuesto. Finalmente, el capítulo 6 resume las conclusiones obtenidas y los problemas que abordaremos como trabajo futuro.

Capítulo 2

Conceptos previos y estado del arte

Este capítulo introduce los siguientes aspectos:

- La arquitectura GPU y la tecnología CUDA para GP-GPU.
- Los trabajos relacionados sobre optimización de aplicaciones paralelas.
- Las librerías Hitmap y Controllers, desarrolladas por el Grupo Trasgo.

2.1. Arquitectura GPU

CUDA [14, 15, 20] es un modelo y una arquitectura de cálculo paralelo de NVIDIA que aprovecha la capacidad de procesamiento de la GPU (unidad de proceso gráfico) para propocionar un incremento del rendimiento de aplicaciones paralelas. CUDA ofrece una plataforma de computación paralela que incluye un compilador y un conjunto de herramientas de desarrollo basadas en el lenguaje C que permite codificar algoritmos para ejecutarlos en GPU de NVIDIA. Existen interfaces de programación que envuelven estas herramientas para poder se utilizadas en otros lenguajes como FORTRAN, Java o Python.

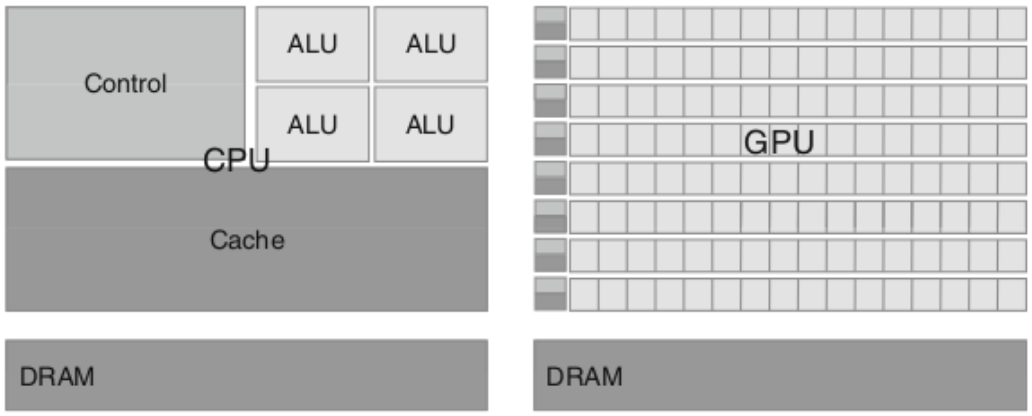


Figura 2.1: Diferencias fundamentales en el diseño de CPU y GPU [15].

La figura 2.1 muestra las diferencias entre los diseños de CPU y GPU. Esta figura representa como la GPU, a diferencia del diseño del CPU, tiene un conjunto de bloques de procesamiento llamados *streaming*

multiprocessors (SM). En la parte derecha de la figura 2.1 cada fila de unidades de proceso (columnas) forma un SM, en la parte izquierda de la fila se encuentran sus unidades de control y de caché. Al igual que en el diseño de CPU, la GPU también tiene una DRAM a la que nos referiremos en adelante como *memoria global*. La figura 2.2 detalla la arquitectura de GPU compatibles con CUDA. Esta figura muestra los componentes de una GPU como son el gestor global de hilos, los *building blocks* (compuestos por uno o varios SM dependiendo de la generación de GPU) y los buses de acceso a memoria global de cada building block. Cada SM (figura 2.3) está compuesto por multiples *streaming procesors* (SP) divididos entre que comparten una unidad de caché de instrucciones, unidad de control lógico, conjunto de registros y unidades de caché de acceso a memoria. Las unidades de procesamiento se diferencian entre: (1) unidades de operaciones de punteros y punto flotante de precisión simple; (2) unidades de operaciones de punto flotante de doble precisión; y (3) unidades de funciones especiales (SFU) de punto flotante de simple precisión. La tabla 2.1 muestra las características de diferentes modelos de GPU NVIDIA.

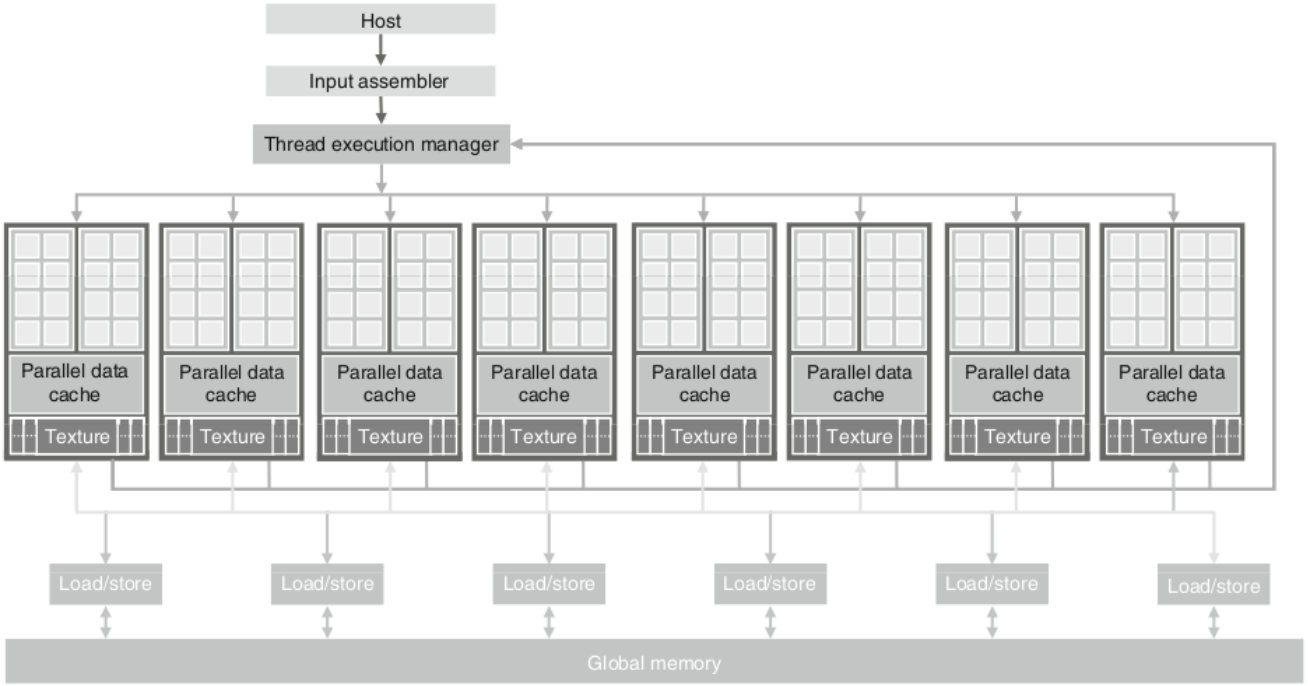


Figura 2.2: La arquitectura de una GPU contiene un ensamblador, controlador de hilos, conjunto de SM y memoria [15].

Modelo	Arquitectura	Cores	Frecuencia	Ancho de banda
8500 GT	Pre-fermi	16 cores	900 Mhz	12.8 GB/sec
9600 GT	Pre-fermi	64 cores	1625 Mhz	57.6 GB/sec
GTX 480	Fermi	448 cores	1215 Mhz	133.9 GB/sec
GTX 680	Kepler	1536 cores	1058 Mhz	192.2 GB/sec
GTX Titan Black	Kepler	2880 cores	980 Mhz	336 GB/sec
Tesla P100	Pascal	3584 cores	1126 Mhz	720 GB/sec
Tesla V100	Tesla	5120 + 640 cores	1531 Mhz	900 GB/sec

Tabla 2.1: Tabla de características de modelos CUDA.

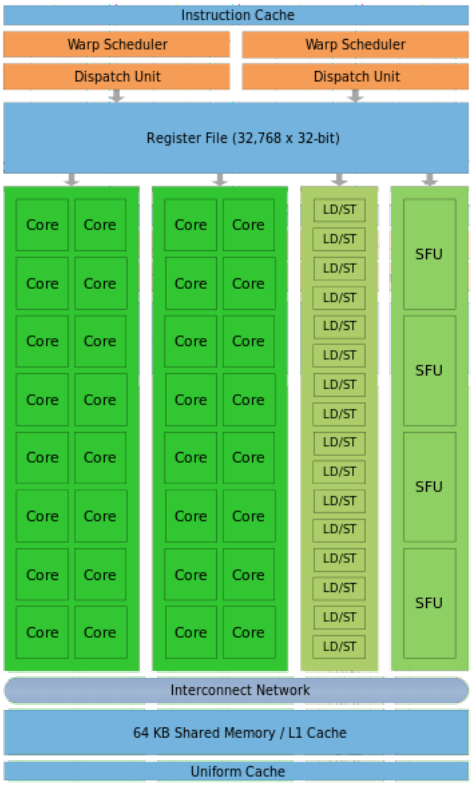


Figura 2.3: Composición de un streaming multiprocessors SM [15].

La figura 2.4 muestra los niveles de abstracción para la programación de GPU con la tecnología CUDA. La tecnología CUDA tiene actualmente las siguientes interfaces de programación de aplicaciones (API): (1) *CUDA driver API* y (2) *CUDA runtime API*. Ambas API están desarrolladas sobre el lenguaje C. CUDA runtime API facilita la programación con la GPU haciendo implícita la inicialización y la gestión de contextos y módulos. En comparación, CUDA driver API ofrece un control de grado fino, especialmente de contextos y cargas de módulos. Un módulo de CUDA agrupa un conjunto de estructuras de datos y funciones dedicadas a cada funcionalidad de CUDA como transferencia de memoria, gestión de errores, gestión de eventos, etc. Un contexto CUDA mantiene la información correspondiente al uso y control de un dispositivo. Esta información incluye la memoria reservada, los módulos cargados, el mapa entre la memoria compartida entre CPU y GPU, etc. El código CUDA codificado mediante CUDA runtime API es una abstracción sobre CUDA driver API y debe ser compilado mediante el compilador de CUDA, *nvcc*.

Un programa CUDA consiste en una o varias fases que son ejecutadas en la máquina anfitriona (*host*) o en un dispositivo como una aceleradora GPU. Un programa CUDA unifica en un único código los códigos correspondientes a estas fases. El compilador nvcc separa cada parte en tiempo de compilación. Ambos código están escritos en ANSI C, el código del dispositivo contiene palabras clave que aporta CUDA runtime API. La tabla 2.2 muestra estas palabras clave. El código del dispositivo tiene funciones paralelizables bajo paralelismo de datos, llamados *kernels*, y las estructuras de datos asociadas a estas.

La figura 2.5 muestra la organización de los hilos de ejecución en bloques (*blocks*) y estos bloques en cuadrículas (*grid*). Esta distribución está limitada por el número máximo de bloques y de hilos posibles. La figura 2.6 muestra un ejemplo de ejecución de un programa CUDA. La configuración de las dimensiones

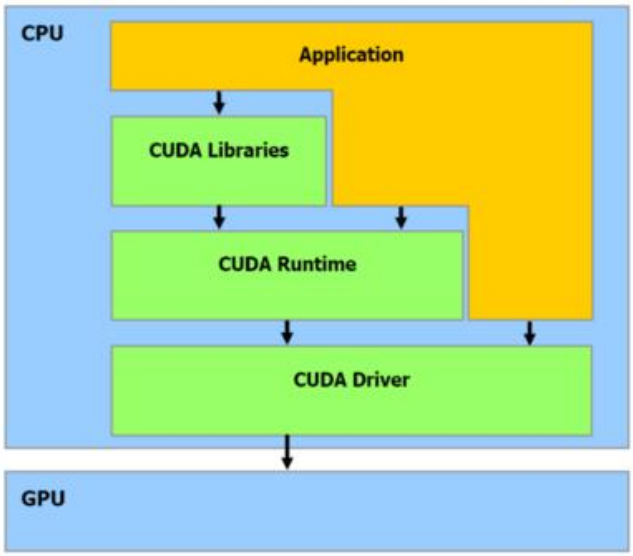


Figura 2.4: Niveles de abstracción de programación sobre GPU NVIDIA mediante CUDA.

Extensión	Ejecutado en	Llamado por
`__device__ type DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ type HostFunc()`	host	host

Tabla 2.2: Tabla de extensiones de modificadores de CUDA [15].

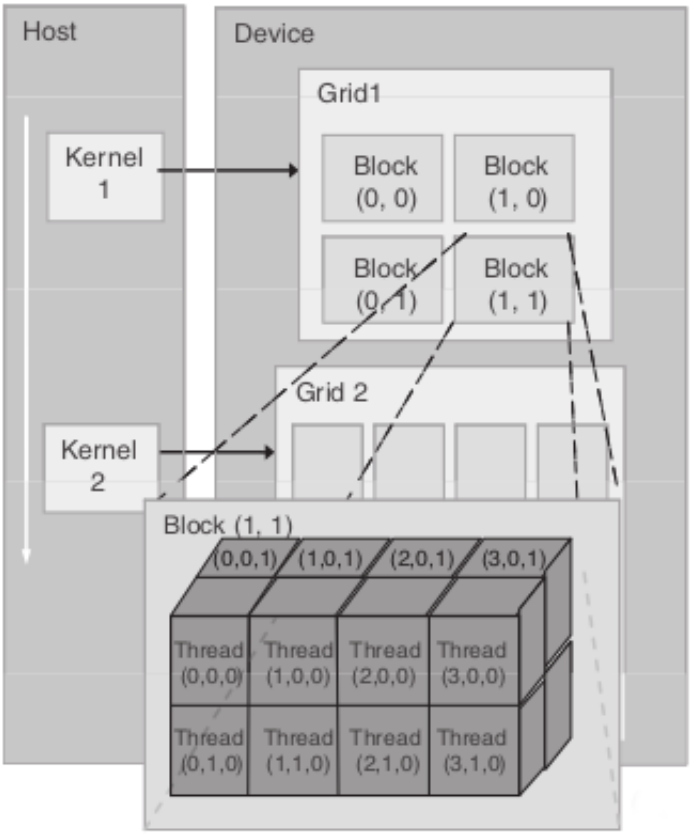


Figura 2.5: Diagrama de organización de CUDA de los hilos de ejecución en bloques y cuadrículas [15].

de las cuadrículas y los bloques se establece individualmente por cada ejecución.

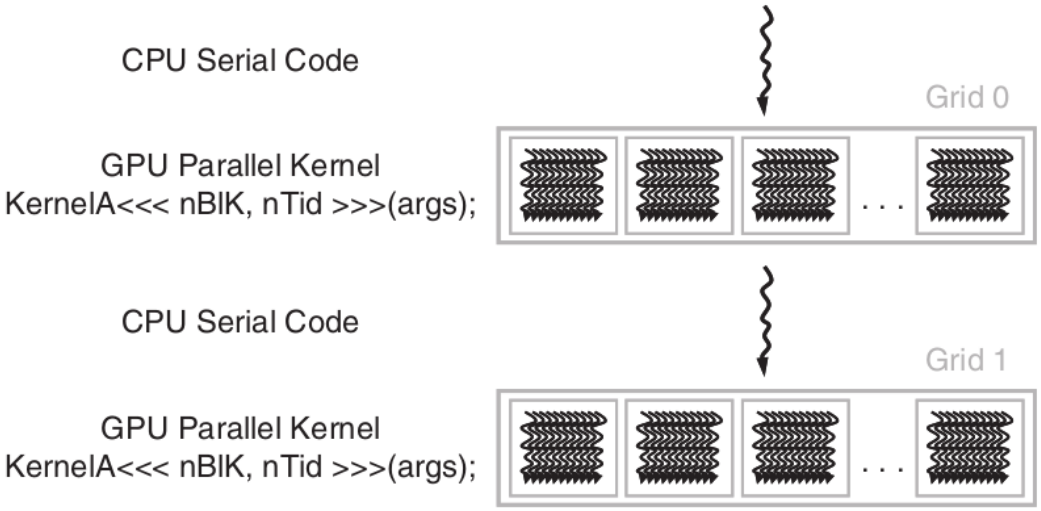
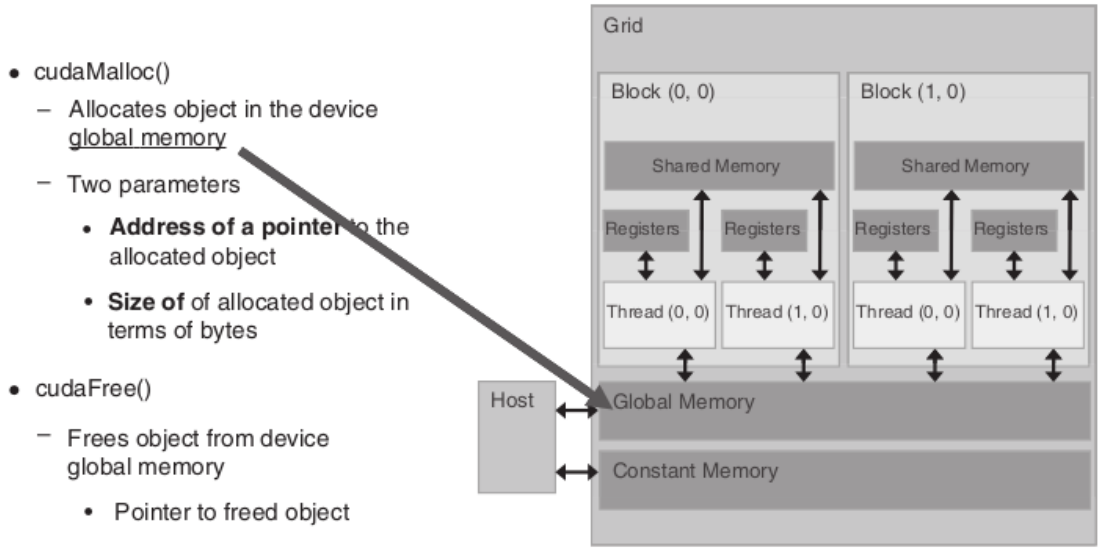


Figura 2.6: Ejecución de un programa CUDA. Se ejecuta inicialmente código secuencia en la máquina host seguida de la ejecución del kernel `KernelA`. Se ejecuta código secuencial den la máquina host. Por último, se ejecuta el kernel `KernelB`. Las expresiones `<<<nBlK, nTid>>>` determinan las dimensiones de los bloques y los hilos por bloque que ejecutan el kernel [15].

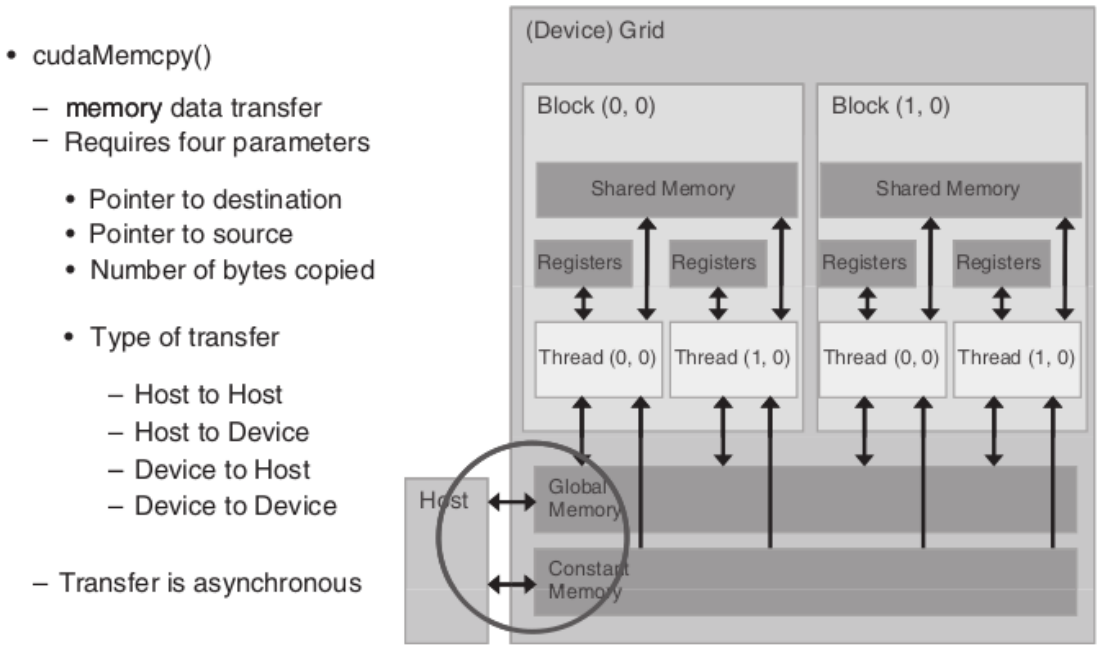
En este modelo de programación, los espacios de memoria de la máquina host y del dispositivo están separadas. Para ejecutar un kernel el programador debe reservar memoria en el dispositivo y transferir los datos correspondientes desde la memoria de la máquina host al dispositivo. La gestión de la memoria del dispositivo se realiza mediante las operaciones `cudaMalloc` y `cudaFree`. La transferencia de memoria se realiza mediante la función `cudaMemcpy`. La figura 2.7 representa las iteraciones que producen estas funciones.

2.2. Trabajos relacionados

Existen tres aproximaciones para gestionar la coordinación de dispositivos heterogéneos. La primera aproximación es diseñar, de forma manual, soluciones utilizando modelos paralelos nativos (incluyendo aquellos provistos por el fabricante o diseñados específicamente para la explotación del conocimiento de la arquitectura) tales como las librerías CUDA [15] y OpenCL [23]. Existen algunos trabajos como los presentados en [13, 21, 22], que utilizan conjuntamente herramientas nativas y los modelos como OpenMP [6] y MPI [11]. El framework CIVL [21] propone un modelo general de concurrencia capaz de representar programas en una gran variedad de dialectos, tales como las librerías previamente mencionadas o la librería Pthreads [18]. La figura 2.8 representa la estructura de este framework CIVL. Este framework ofrece cuatro front-end para cada uno de los dialectos, y un back-end que verifica diferentes propiedades del programa como la aparición de deadlocks, condiciones de carrera, fugas de memoria, etc. Esta verificación se realiza a través del uso de un modelo de comprobación y una ejecución simbólca del programa. Sin embargo, CIVL no permite una optimización adicional dependiente de la arquitectura. También hace necesaria la implementación manual del modelo de concurrencia que tiene la aplicación.



(a) Funciones de CUDA runtime API para la gestión de memoria.



(b) Funciones de CUDA runtime API para la transferencia de memoria.

Figura 2.7: Funciones de CUDA runtime API para las operaciones de memoria [15].

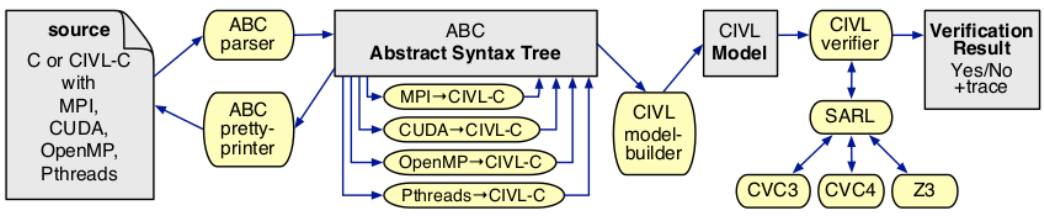


Figura 2.8: Framework CIVL [21].

El modelo dCUDA [13] (distributed CUDA) ofrece un modelo conjunto de MPI y CUDA para realizar comunicaciones directas entre GPU de un cluster. Este modelo permite el solapamiento de comunicación y computación mediante el acceso directo a memoria (*DMA*, direct memory access) entre dispositivos mediante funcionalidades de MPI como muestra la figura 2.9. Las comunicaciones de dCUDA se utilizan dentro del código de un kernel CUDA; al inicio de un kernel se crean las ventanas de memoria por las que se realizaran estas comunicaciones. En cambio, este modelo no supone mejoras de rendimiento al usarse sobre un único dispositivo. También supone cambiar el código interno del funcionamiento de CUDA, tal y como se describe en el trabajo, entrando en conflicto con algunos principios de esta tecnología.

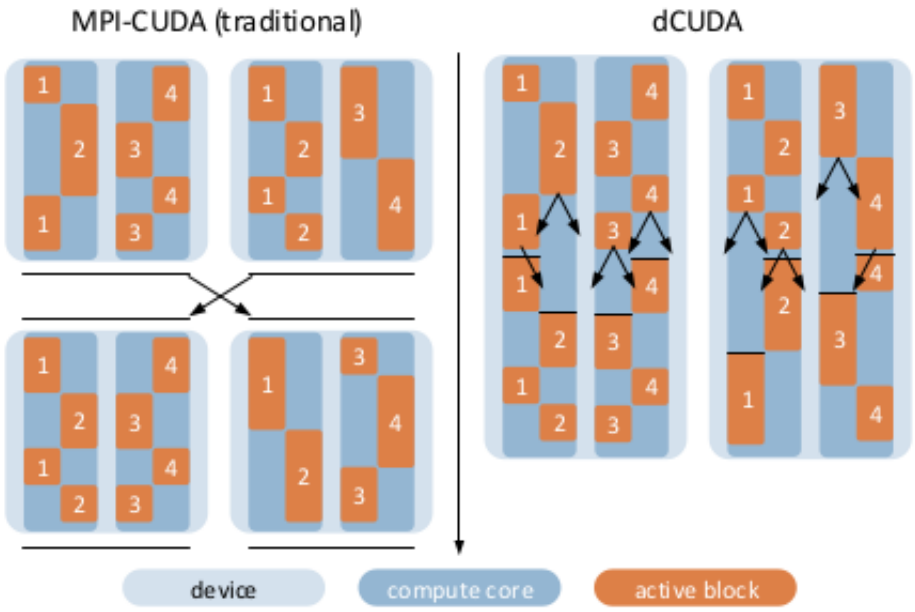
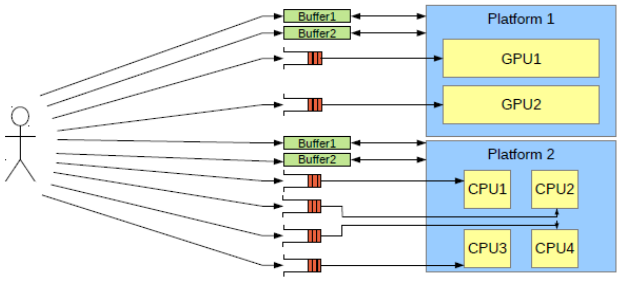


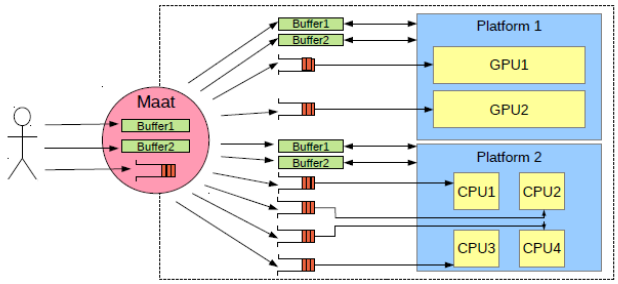
Figura 2.9: Comparación de comunicaciones de bloques entre dispositivos mediante MPI-CUDA y dCUDA. La solución MPI-CUDA supone la sincronización global de los dispositivos para la comunicaciones entre bloques, realizándose esta gestión desde el host. La solución con dCUDA muestra como estas comunicaciones se realizan sin necesidad de un control desde el host [13].

Otra aproximación más abstracta se encuentra en el uso de herramientas, librerías, y estrategias orientadas en la ejecución paralela de bucles, de forma automática, en dispositivos heterogéneos; tales como LogFit [24] y Maat [19]. Estas propuestas dividen las iteraciones de un bucle en tareas utilizando técnicas para movimiento asíncrono de datos, también ejecutan dichas tareas en los diferentes dispositivos. LogFit planifica los tamaños de computación según un estudio anterior [5] que determina como el rendimiento de la GPU sige una curva logarítmica respecto al tamaño de partición: $y = a\ln(x) + b$. Según esta relación el planificador LogFit ajusta el tamaño inicial de la partición. Después adapta este tamaño en itervalos de tiempo, ajustándolo según el rendimiento obtenido en cada intervalo. En cambio, Maat es una librería de OpenCL que permite la ejecución eficiente de un único kernel utilizando todos los dispositivos disponibles. La librería Maat ofrece una abstracción sobre la estructuras de gestión que ofrece OpenCL para cada dispositivo usado. Esta librería ofrece: (1) un super contexto, (2) un super buffer, (3) un super kernel. Maat gestiona el balanceo de carga se una forma similar a la de OpenMP ofreciendo tres métodos: (a) método estático, un reparto inicial de los trabajos entre los nodos; (b) método dinámico, haciendo una división inicial en paquetes de trabajos de tamaño fijo que se reparte mediante un hilo o proceso maestro; y (c) método guiado, similar al método anterior aunque los tamaños de paquetes de trabajos se reducen

en el progreso de la ejecución.



(a) Gestión utilizando OpenCL



(b) Gestión utilizando Maat

Figura 2.10: Maat: Tareas relacionadas con la gestión de un sistema heterogéneo de ejemplo [19].

Sin embargo, estos trabajos no soportan la gestión de árboles de dependencias entre tareas generadas por bucles anidados. El uso de estas abstracciones no supone una mejora en la eficiencia del uso de los recursos del dispositivo en todos los casos.

2.2.1. Trabajo desarrollado por Grupo Trasgo

Esta sección introduce conceptos básicos de las librerías Hitmap [12] y Controllers [17]. Además, se describe la arquitectura y las operaciones básicas de transferencia de datos de la implementación previa de Controllers a través de un código que actúa como ejemplo.

Hitmap

La librería Hitmap ofrece ténicas de particionado y mapeo automático de datos, de forma eficiente y configurable, en tiempo de ejecución. Esta librería define una interfaz de abstracción y un sistema de plug-in que encapsula técnicas regulares e irregulares, ayudando a generar código independientemente de la función de mapeo seleccionado. Hitmap soporta el particionado (*tiling*) de array distribuciones dispersas o compactas, este particionado permite la impolementación de paralelismo de datos y tareas según el modelo SPMD. Esta librería ofrece funcionalidades para crear, manipular, distribuir y comunicar estas particiones (*tiles*) y jerarquía de pariciones.

La librería Hitmap introduce los siguientes conceptos:

Signature: Una *signature* S se define como una terna que representa un subespacio de indices sobre

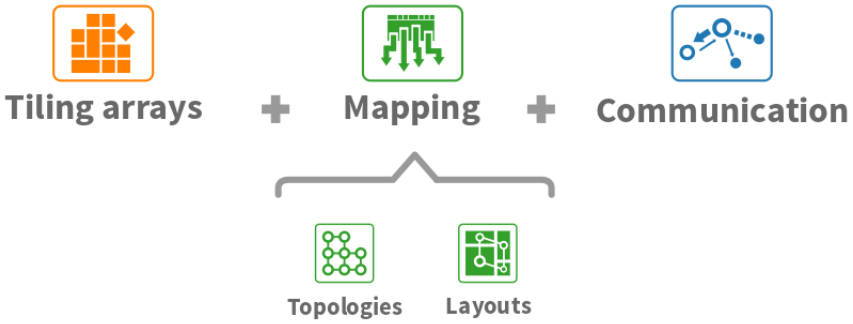


Figura 2.11: Funcionalidades de la librería Hitmap.

un array unidimensional. Una signature sigue la notación de Fortran90 or MATLAB de selección de índices de un array. La cardinalidad de una signature es el número de índices diferentes del dominio.

$$S \in \text{Signature} = (\text{begin} : \text{end} : \text{stride})$$
$$\text{Card}(s \in \text{Signature}) = \lfloor (s.end - s.begin)/s.stride \rfloor$$

Shapes: Una *shape* h es una n-tupla de signatures. Representa una selección de un subespacio de los índeces de un array con dominio multidimensional. La cardinalidad de una shape es el número de diferentes combinaciones de índices del dominio.

$$h \in \text{Shape} = (S_0, S_1, S_2, \ldots, S_{n-1})$$
$$\text{Card}(h \in \text{Shape}) = \prod_{i=0}^{n-1} \text{Card}(S_i)$$

Tile: Un *tile* es un array n-dimensional. Su dominio es definido por un shape, y tiene un tiene un número de elementos de un tipo (*type*) dado.

$$\text{Tile}_{h \in \text{Shape}} : (S_0 \times S_1 \times S_2 \times \ldots S_{n-1}) \rightarrow \texttt{<type>}$$

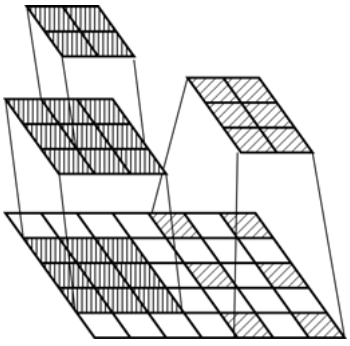


Figura 2.12: Creación de tiles de un array original [12].

Hitmap ofrece tres conjuntos de funcionalidades:

Funciones de tiling: Definición y manipulación de arrays y tiles. Estas funciones pueden utilizarse de forma independiente. Estas funciones mejoran el rendimiento al acceder a los datos en código secuencial al igual que al generar menualmente distribuciones de datos para ejecuciones paralelas. La figura 2.12 muesta un ejemplo de jerarquía de tiles.

Funciones de mapeo: Funciones de distribución y disposición (*layout*) de datos que particionan de forma automática los dominios de los arrays en tiles, dependiendo de la topología virtual seleccionada.

Funciones de comunicación: Creación de patrones de comunicación para jerarquías de tiles distribuidas. Estas funciones son una abstracción de comunicaciones sobre un modelo de paso de mensajes entre diferentes procesos. Se pueden crear patrones dependientes de la información sobre el mapeo de un tile.

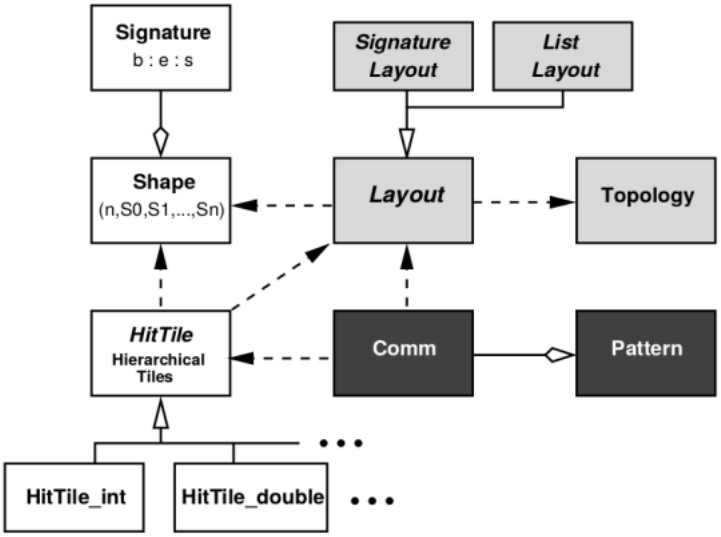


Figura 2.13: Diagrama UML de la arquitectura de la librería Hitmap [12].

Controllers

El modelo *Controllers* ofrece una metodología de programación sistemática con importantes características, tales como:

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

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

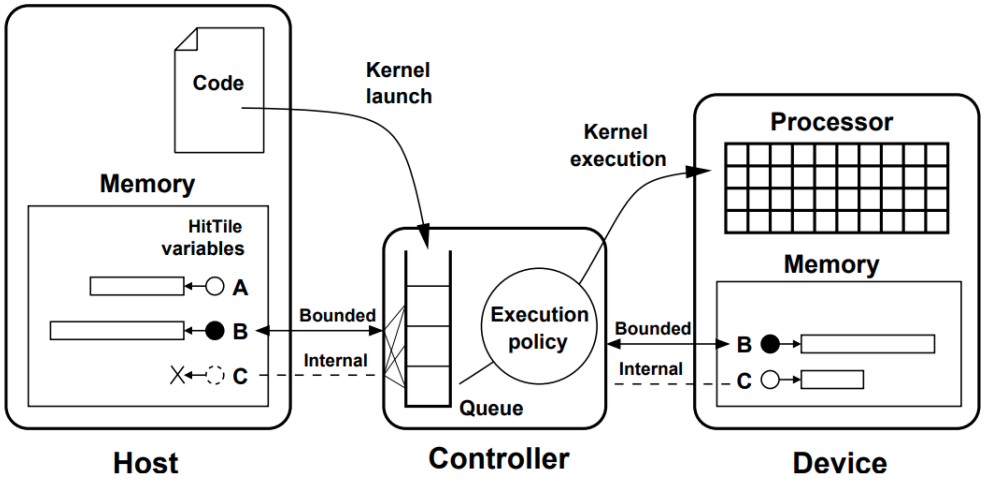


Figura 2.14: Diagrama de la arquitectura del modelo Controllers. Los lanzamientos de kernel son encolados. La entidad de Controller controla la ejecución de los kernel enclados y las transferencias de datos entre los espacios de memoria. En la figura, la variable A en el host no está asignada al Controller. La variable B si lo está y tiene una imagen de los datos en la memoria del dispositivo. La variable C es una variable interna (*internal*), definida en el host, pero alojada únicamente en el dispositivo.

La figura 2.15 desarrolla un ejemplo de uso de las librerías Hitmap y Controllers. Esta figura muestra el código de una implementación del método Jacobi-2D para un número determinado de iteraciones. En cada iteración las estructuras de datos son recuperadas por la máquina host desde el dispositivo GPU. En las líneas 2 y 9 de la figura 2.15 se muestra la definición de dos kernel genérico. Un kernel especifica el algoritmo a ejecutar mientras que Controller, automáticamente lo adaptará para que pueda ser ejecutado en cualquiera de los dispositivos soportados (GPU, XeonPhi o procesadores multi-core). La declaración de los kernel contiene información de cada uno de los parámetros que va a tener mediante un conjunto de ternas. Cada terna incluye tipo, nombre y rol (parámetro de entrada o de salida). Para implementar un kernel dentro de una aplicación es necesario seguir la siguiente estructura de desarrollo:

1. creación de la entidad de Controller asociada a un dispositivo concreto (líneas 20-21);
2. asociar cada estructura de datos *HitTile* a una instancia del Controller (líneas 29-30);
3. asociar la ejecución de un kernel con la misma instancia de Controllers que se utilizó para sus parámetros *HitTile* (líneas 36-37);
4. liberar las estructuras de datos *HitTile* de la instancia del Controller (líneas 43-44);
5. destrucción de la instancia de Controller (línea 45).

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

```
1  /* Matrix copy movement: Generic kernel code for any type of device */
2  KERNEL(copy, 2, OUT, HitTile_float, dst,  IN,  HitTile_float, src)
3  {
4     int x = thread.x + 1; int y = thread.y + 1;
5     hit( dst, x, y ) = hit( src, x, y );
6  }
7
8  /* Jacobi iteration: Generic kernel code for any type of device */
9  KERNEL(jacobi, 2, OUT, HitTile_float, A,  IN,  HitTile_float, B )
10 {
11    int x = thread.x + 1; int y = thread.y + 1;
12    hit( A, x, y ) = ( hit( B, x-1, y ) + hit( B, x+1, y ) +
13                   hit( B, x, y-1 ) + hit( B, x, y+1 ) ) / 4;
14 }
15
16 /* Host program using the Controller library */
17 int main()
18 {
19   /* Stage 1: Controller creation, associated to the first GPU in the system */
20   Cntrl comm;
21   CntrlCreate(&comm, CNTRL_GPU, 0);
22
23   /* Stage 2: Data structures creation and initialization */
24   HitTile_float A = hitTile( float, 2, SIZE, SIZE );
25   HitTile_float B = hitTile( float, 2, SIZE, SIZE );
26   initMatrices(&A);
27
28   /* Stage 3: Data structures attachment: Associate to the Controller context */
29   CntrlAttach(&comm, &A);
30   CntrlCreateInternal(&comm, &B);
31
32   /* Stage 4: Kernel launchings with a number of logical fine-grain threads */
33   Thread threadsSpace = ThreadInit( SIZE - 1, SIZE - 1);
34   for( int iter=0; iter<NUM_ITERS; iter++ )
35   {
36    CntrlLaunch(comm, copy, threadsSpace, 2, &B, &A);
37    CntrlLaunch(comm, jacobi, threadsSpace, 2, &A, &B);
38    CntrlMoveFrom(comm, &A);
39    usePartialResultInHost(A);
40   }
41
42   /* Stage 5: Data structures detachment, and free device resources */
43   CntrlDetach(&comm, &A);
44   CntrlDestroyInternal(&comm, &B);
45   CntrlDestroy(&comm);
46 }
```

Figura 2.15: Definición de kernel y programa principal de una implementación stencil del método Jacobi usando la librería Controllers.

2.3. Conclusiones del capítulo

CUDA ofrece una arquitectura que facilita la programación GP-GPU sobre dispositivos GPU de NVIDIA. El modelo de programación separa los espacios de computación y comunicación entre la máquina anfitriona o host y el dispositivo. La programación de algoritmos para GPU se dividen en operaciones de gestión de memoria y operaciones de computación o kernels.

Los trabajos presentados optimizan el rendimiento de aplicaciones paralelas GPU, mediante diferentes técnicas de solapamiento de comunicación y computación. Sin embargo, estas optimizaciones se enfocan en la optimización en cluster con varias GPU en más de un nodo, o en la planificación de iteraciones no permite el uso, a mayores, de optimizaciones propias de la tecnología subyacente.

Hitmap y Controllers ofrecen capas de abstracción sobre el uso de estructuras de datos distribuidas, y la implementación de código paralelo portable independiente del hardware sobre el que va a ser ejecutado. Este trabajo extiende el modelo de Controller ofreciendo un sistema transparente para el usuario de solapamiento de tareas de comunicación y computación. Este sistema se basa en el análisis de la información de los parámetros de un kernel, y la posibilidad de solapar su computación con peticiones (*requests*) de comunicación.

Capítulo 3

Descripción de solución

Este capítulo introduce los siguientes aspectos:

- Una extensión del modelo Controllers que permite el solapamiento automático de operaciones de comunicación y computación.
- El límite teórico máximo de aceleración que se puede obtener mediante el solapamiento de operaciones de comunicación y computación.

3.1. Propuesta de solución

El modelo que se propone, en este capítulo, mejora el uso de los recursos paralelos de un dispositivo mediante el solapamiento de tareas de comunicación y computación. El modelo aporta una política de gestión que permite detectar las operaciones que puedan solaparse sin riesgo de que aparezcan problemas de consistencia de los datos involucrados.

Esta política de gestión de operaciones es transparente para el usuario, sin que deba modificar los programas ya codificados según el modelo Controllers [17]. Esta política también es independiente del modelo programación paralela que se siga, es decir, dicha política es usable sobre distintos modelos de paralelismo y sobre distintas tecnologías, como clusters many-core, FPGA, co-procesadores y aceleradoras GPU.

3.2. Descripción de Controller

La figura 3.1 representa la nueva arquitectura propuesta para librería de Controllers. Dicha arquitectura contiene una cola de peticiones, que incluyen la secuencia de lanzamientos de kernel y las peticiones de transferencia de datos. En el modelo previo de Controllers existen las siguientes peticiones de transferencia de datos de: (1) *attach*, enlace de una estructura de datos a la instancia de Controller, con reserva de recursos y envío de la estructura de datos del anfitrión al dispositivo; (2) y *detach*, recuperación de una estructura de datos, y liberación de recursos de la instancia del Controller. Esta última implica una

operación bloqueadora hasta que los datos de la transferencia estén disponibles y los recursos se hayan liberado. El modelo propuesto añade dos operaciones de transferencia de datos. Estas nuevas operaciones ayudan en la actualización de los datos entre su estado en el espacio de memoria de la máquina host y la de la aceleradoras.

En adelante, a las operaciones contenidas en la cola de peticiones se les denomina *tareas*.

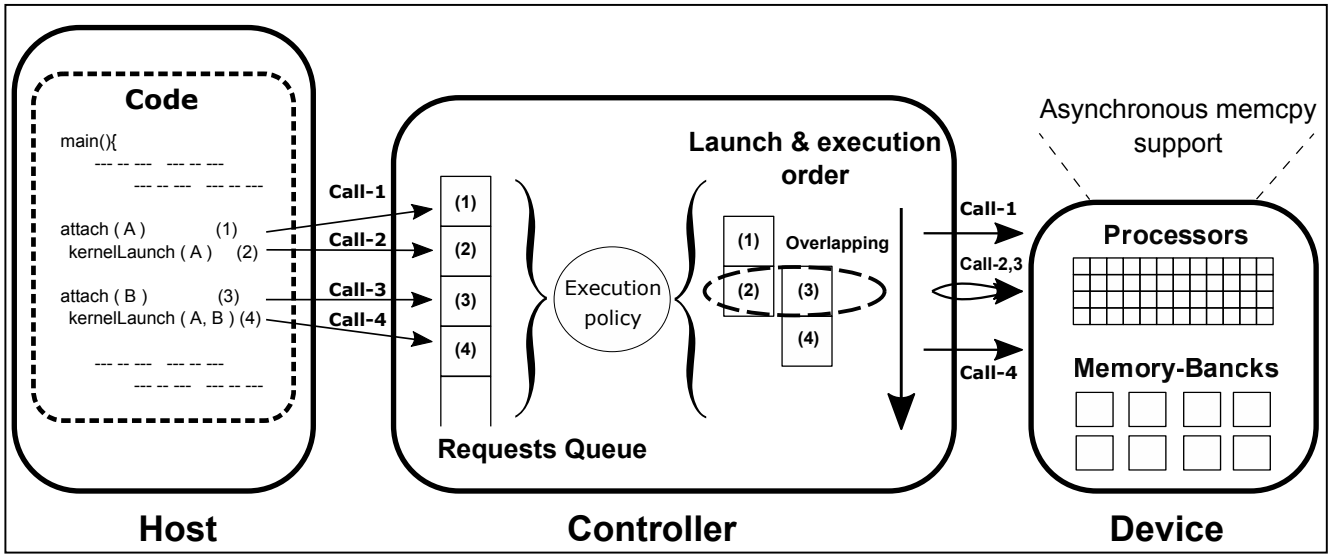


Figura 3.1: Diagrama del modelo de arquitectura propuesta. Las peticiones de movimientos asíncronas son añadidas a una cola. La entidad del Controlador será la encargada de la ejecución de estas transferencias y las ejecuciones de los kernels.

3.3. Modelo de tareas asíncronas

Este apartado describe como se produce el solapamiento de tareas de comunicación y computación, en el modelo propuesto, mediante el caso que muestra la figura 3.1. En este ejemplo, el usuario implementa un programa paralelo usando las operaciones habituales de Controller (parte izquierda de figura 3.1). Este ejemplo contiene cuatro principales tareas, dos de ellas implican transferencias, y las otras dos, lanzamientos de kernel. La instancia de Controller almacena en la cola las tareas especificadas por el usuario, siguiendo el orden de inclusión. El primer kernel tiene A como entrada. Por lo tanto, la ejecución de este kernel necesita ser sincronizada con el final del movimiento de A al dispositivo. La estructura de datos B es usada como entrada del segundo kernel, pero no es salida del primer kernel. Por consiguiente, la transferencia de B se puede solapar con la ejecución del primer kernel (parte derecha del centro de la figura 3.1). Sin embargo, la ejecución del segundo kernel tiene que sincronizarse con el final de la transferencia al dispositivo de B.

Este ejemplo nos sirve para introducir una primera aproximación sobre aquellas tareas pueden solaparse manteniendo la equivalencia secuencial:

a) una tarea de lanzamiento de kernel debe ser sincronizada, esperando a cualquier comunicación que

afecte a los parámetros del kernel;
b) las tareas de lanzamiento de kernel son ejecutadas de forma sincrónica entre sí, manteniendo el orden en el que aparecen en la cola;
c) una tarea de comunicación de datos de actualización puede solaparse con ejecuciones previas de kernels siempre que no sea usada como parámetro de este kernel;
d) se debe permitir que el host (el programa principal) pueda esperar por el final de una transferencia de una estructura de datos desde el dispositivo.

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

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

En el ejemplo de la figura 3.1, el Controller puede realizar las tareas (2) y (3) de forma concurrente en el dispositivo. La tarea (4) se despacha cuando las variables A y B han sido transferidas de forma completa a la memoria global del dispositivo (fin de comunicaciones previas), y el kernel previo haya finalizado.

3.4. Máxima mejora teórica de rendimiento

Esta sección muestra el cálculo de la máxima mejora teórica de rendimiento según la ley de Amdahl. Esta mejora de rendimiento se supone sobre un modelo secuencial puro, como el modelo previo de Controllers donde la ejecución de las tareas asignadas se producía por una única cola de ejecución.

3.4.1. Ley de Amdahl

La ley de Amdahl es un modelo matemático que describe la relación entre la aceleración esperada de la implementación paralela de un algoritmo y la implementación secuencial del mismo algoritmo. La ley de Amdahl permite calcular la aceleración S que se puede alcanzar a partir de las modificaciones de una porción P, según la ecuación 3.1.

$$\frac{1}{(1-P)+\frac{P}{S}} \tag{3.1}$$

3.4.2. Aplicación de la ley de Amdahl

Con una política sincrónica pura, todas las tareas para ejecutar kernels (computación) o movimientos de datos (comunicación) se evalúan de forma secuencial y el tiempo final de ejecución de la aplicación es la suma de los tiempos de ejecución de las tareas. Considerando T_i el tiempo de ejecución de una tarea de computación, y C_i el tiempo de ejecución de una comunicación. El tiempo final de ejecución es $\sum_i T_i + \sum_j C_j$.

En nuevo modelo propuesto, la computación todavía se ejecuta de forma sincrónica entre sí, y consideramos *la parte secuencia* de toda la ejecución. Los tiempos de las tareas de comunicación pueden solaparse con la computación. Muchas tareas asíncronas de comunicación pueden iniciarse al mismo tiempo. Sin embargo, debido a la limitación natural de los canales de comunicación entre la máquina host y la aceleradora, típicamente el bus PCIe, los tiempos de transferencia son mayormente secuenciales igualmente. Para simplificar no hemos tomado en cuenta las mejoras que el driver del dispositivo pueda aplicar en los buffers de comunicación.

Por lo tanto, en un escenario ideal, donde los tiempos de comunicación y computación pueden solaparse perfectamente, el tiempo de ejecución final es el máximo entre $\sum_i T_i$ y $\sum_j C_j$. El mejor escenario posible ocurre cuando los dos tiempos tiene valores similares según $\psi = \sum_i T_i = \sum_j C_j$. Asumiendo una perfecta simultaneidad entre computación y comunicaciones, el máximo teórico de mejora de rendimiento está limitada la ecuación 3.2.

$$S = \frac{\sum_i T_i + \sum_j C_j}{\text{máx}(\sum_i T_i, \sum_j C_j)} \tag{3.2}$$

O de forma simplificada en la ecuación 3.3:

$$S = \frac{\psi + \psi}{\text{máx}(\psi, \psi)} = 2 \tag{3.3}$$

3.5. Conclusiones del capítulo

El modelo propuesto ofrece un sistema automático y transparente para el usuario de solapamiento de operaciones de comunicación y computación basado en información de las operaciones dada por el usuario. Este modelo propuesto aprovecha la información para encontrar aquellas tareas que puedan solaparse sin riesgo de que se produzca inconsistencia de memoria. Este solapamiento de operaciones presenta según la aplicación de la ley de Amdahl una mejora máxima de $\times 2$, es decir, supondría una reducción del 50 % en el tiempo total de ejecución de una aplicación. Para alcanzar esta situación los tiempos de comunicación y computación que se solapen deben ser parecidos siempre, obteniendo la mejora óptima cuando $\psi = \sum_i T_i = \sum_j C_j$. Esta mejora se consigue siempre y cuando las tareas no tengan dependencias entre sí y deban secuencializarse.

Capítulo 4

Implementación

Este capítulo introduce los siguientes aspectos:

- Introducción al solapamiento de tareas de comunicación y computación mediante funcionalidades CUDA.
- Descripción de las aproximaciones que ha tenido la implementación del prototipo para el modelo propuesto.
- Implementación de las llamadas asíncronas de controladores a la aceleradora del prototipo del modelo.
- Detalle de las políticas para las dependencias de datos entre comunicaciones y computación.

4.1. Solapamiento de trabajos con CUDA

La librería Controllers hace uso de la tecnología CUDA para el código determinado que va a ser ejecutado sobre aceleradoras GPU de NVIDIA. CUDA ofrece una funcionalidad llamada *stream*, que se describe en [20] como una cola FIFO de operaciones GPU que son ejecutadas en un orden específico. Algunos modelos de aceleradoras de NVIDIA soportan el uso de varios streams en un dispositivo. El uso de varios streams de ejecución permite el solapamiento de operaciones como se muestra en la figura 4.1. Las aceleradoras GPU de NVIDIA tienen un stream por defecto por el que se realizan todas las operaciones realizadas mediante la librería CUDA. La librería de Controllers no utiliza este stream con la intención de tener un mayor control sobre la ejecución de cada tarea, así cada instancia de Controller tiene un stream propio.

Para acelerar la transferencia asíncronas de datos entre el dispositivo y el host en el uso de varios streams, el modelo CUDA necesita el uso de *page-locked memory* o *pinned memory* (memoria bloqueada o fija) [20]. Esta memoria tiene la siguiente propiedad: El sistema operativo garantiza que nunca se transferirá esta memoria al disco, lo que asegura que esta memoria se encontrará en memoria física. El corolario de esta propiedad es que es seguro para el sistema operativo permitir el acceso a una aplicación a una dirección física de memoria, ya que el buffer de transferencia no va a ser desalojado o movido. Este tipo de memoria permite que el dispositivo utilice transferencias de acceso directo a memoria (*DMA*, *Direct*

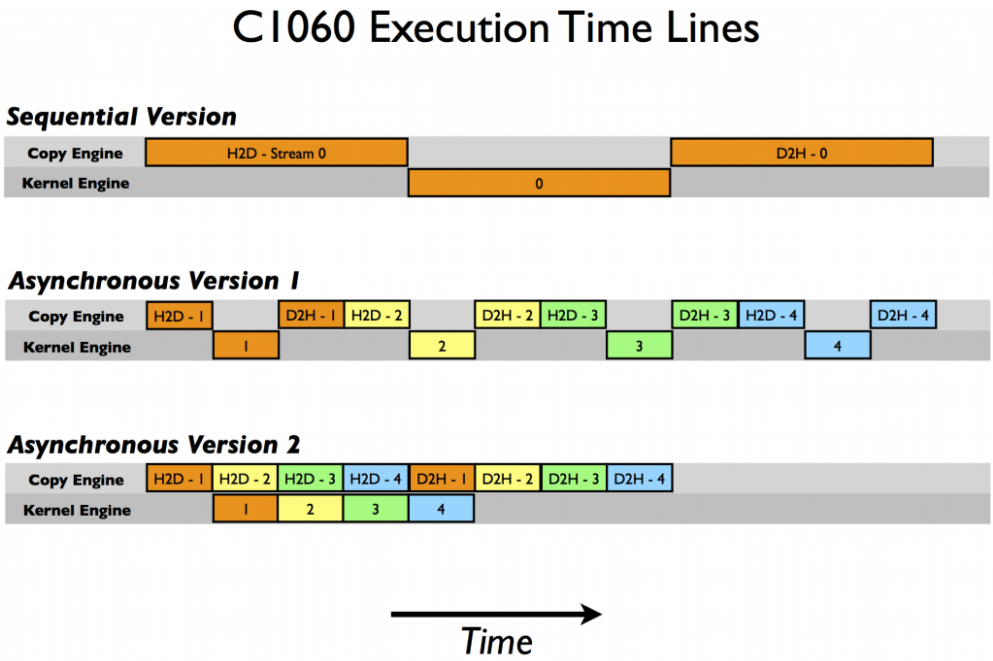


Figura 4.1: Ejemplo de solapamiento de comunicación y computación con streams CUDA [3]. Las filas representan los canales de ejecución (*Kernel Engine*) y transferencia (*Copy Engine*) que tiene un dispositivo GPU. Cada color de bloque representa un stream distinto. Los etiquetas de los bloques tienen los siguientes significados: i (bloque numerado), operaciones de lanzamiento de kernel por stream i; *H2D-i*, operación de transferencia de host al dispositivo por stream i; *D2H-i*, operación de transferencia de dispositivo al host por stream i. La versión **Asynchronous Version I** divide en tareas más pequeñas las operaciones de la versión base **Sequential Version** ejecutándolas de forma secuencial. La versión **Asynchronous Version II** reorganiza la ejecución de estas tareas de forma que puedan solaparse, consiguiendo una reducción del tiempo total de ejecución.

Memory Access) entre el host y el dispositivo. Estas transferencias DMA se producen sin la intervención de la CPU. A diferencia de la reserva de memoria física con C mediante la función `malloc`, CUDA permite la reserva de memoria pinned mediante la función `cudaHostAlloc`. Esta memoria se libera posteriormente mediante la función `cudaFreeHost`.

En la arquitectura previa de Controllers, la cola de tareas preparadas (parte derecha de figura 3.1) se implementa mediante el stream de la instancia, al que se asignarán las transferencias de datos y los lanzamientos de kernels. Para permitir el solapamiento de comunicaciones y computación, estas operaciones deben producirse por diferentes streams del dispositivo. Por ello, en la implementación del prototipo del modelo asíncrono del Controller, se le asigna un stream propio a cada estructura de datos al asociarla con una instancia de Controller. Por este stream de la estructura se ejecutarán sus tareas de transferencia. Las tareas de computación se realizan por el stream de la instancia de Controller. Es necesario, liberar posteriormente este stream al desligar la estructura de la instancia de Controller (operación de detach). Además, la implementación de prototipo utilizará memoria pinned para la transferencias entre en host y el dispositivo según necesita el modelo CUDA para ejecuciones por diferentes streams.

4.2. Desarrollo de las diferentes aproximaciones

Hemos iterado sobre diferentes aproximaciones hasta alcanzar el prototipo final que permite validar el modelo propuesto. Esta sección presenta las aproximaciones iniciales; por cada aproximación describimos las mejoras sobre la versión previa y los problemas encontrados por los que iteramos a la siguiente aproximación. El lenguaje elegido para la implementación es C, ya que es el lenguaje utilizado en las librerías de Hitmap y Controllers.

Las diferentes aproximaciones reducen el tiempo de ejecución mediante el solapamiento de tareas de comunicación y computación según las especificaciones del modelo propuesto. Las aproximaciones difieren entre sí en la implementación de la gestión de dependencias:

- Una primera aproximación gestiona las dependencias entre tareas mediante esperas por los streams que bloquean la extracción de tareas de la cola de una instancia de Controller, evitando la evaluación de nuevas tareas que suponga problemas de consistencia en los datos.
- Una segunda aproximación establece las esperas directamente entre los diferentes streams mediante estructuras de control CUDA. Estas esperas se añaden en tiempo de ejecución al extraer una tarea de la cola, y las gestiona CUDA, de forma interna, en el dispositivo.
- En una última aproximación, se añaden estructuras propias a las librerías Hitmap y Controllers para la gestión interna de las esperas y la ejecución de las tareas. Las esperas se establecen en la evaluación de las tareas siempre que haya dependencias entre ellas, y se resuelven al final de la ejecución de la tarea que corresponda.

4.2.1. Primera aproximación: bloqueo de la ejecución del hilo principal

La primera aproximación resuelve las dependencias entre tareas esperando mediante la función CUDA: `cudaStreamSynchronize(cudaStream_t)`. Esta función añade un punto de sincronización en la cola de

tareas del stream dado y bloquea el hilo de ejecución desde la que es llamada en la máquina host, hasta que el stream termina las ejecuciones pendientes previas al punto de sincronización. En una instancia de Controller, los puntos de bloqueo (llamadas a la función) se establecen en la evaluación de tareas, esperando por aquellas ejecuciones en stream diferentes con los que se tenga alguna dependencia de datos. Esta espera bloquea la extracción de nuevas tareas. En esta aproximación se han encontrado las siguientes limitaciones:

1) La primera limitación de esta aproximación es el bloqueo de la ejecución que extrae de la cola de tareas de un Controller (véase figura 3.1 en sección 3.2). Este bloqueo impide la evaluación de nuevas tareas, y por lo tanto posibles solapamientos con tareas posteriores de la cola.
2) La segunda limitación se debe a que no es posible gestionar, de manera independiente, las esperas que se deban producir en el ejecución principal del programa (parte derecha de la figura 3.1) y las de un Controller (parte central de la figura 3.1). Dado que la ejecución del programa principal no tiene un punto de sincronización hasta una operación de detach.
3) La tercera limitación se encuentra en que solo es posible esperar por la ejecución completa de un stream y no de una operación en particular. Esto significa que para esperar por la ejecución de una tarea se debe establecer el punto de sincronización después del lanzamiento de esta por un stream. En caso contrario, se realizará una espera por todas las operaciones pendientes de un stream previas al punto de sincronización.

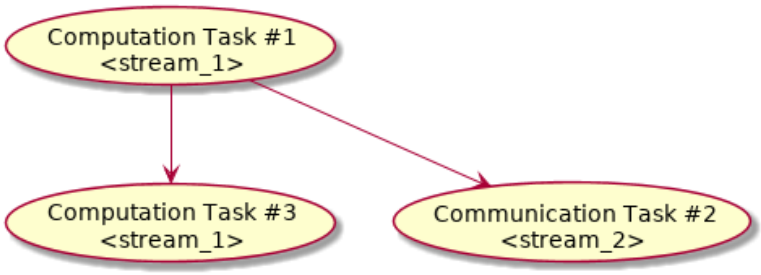
La figura 4.2 muestra los efectos de esta última limitación, y propone un escenario de ejemplo con dos casos de estudio. La figura 4.2a muestra las dependencias entre las tres tareas del escenario; dos de ellas (tareas 1 y 3) se ejecutan sobre el stream 1 y la tarea 2 sobre el stream 2; la tarea 2 tiene una dependencia de ejecución de la tarea 1. La figura 4.2b muestra el caso donde se produce el procedimiento correcto, la llamada de sincronización (espera por la finalización de la tarea 1) se realiza justo después del lanzamiento de la tarea 1, lo que supone que la tarea 2 se pueda ejecutar de forma segura. Por otra parte, la figura 4.2b expone el caso problemático, la función de sincronización –con la tarea 1– se llama después del lanzamiento de la tarea 3, haciendo que la tarea 2 tenga que esperar al final de la ejecución de todas las tareas del stream.

4.2.2. Segunda aproximación: gestión de esperas sin bloqueo del hilo de ejecución

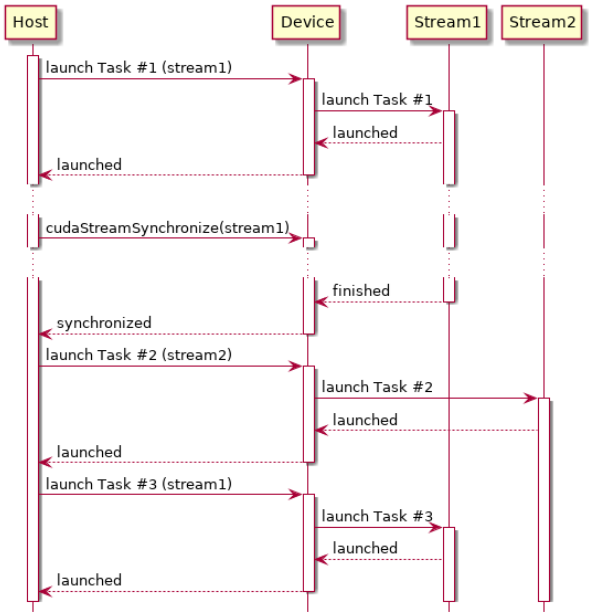
CUDA ofrece estructuras de control para gestionar las dependencias de datos entre ejecuciones de stream. Estas funcionalidades CUDA son los eventos CUDA o `cudaEvent_t`. Los eventos CUDA permiten establecer puntos de sincronización entre varios streams en la ejecución de una aplicación paralela. La gestión de las relaciones entre streams y eventos CUDA se realiza mediante las siguientes funciones:

`cudaEventRecord(cudaEvent_t, cudaStream_t)`: Establece un punto de notificación dentro de la cola de operaciones del stream dado.

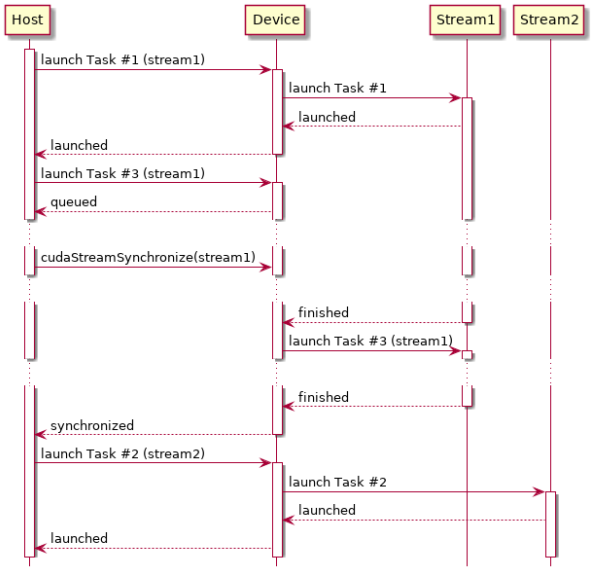
`cudaStreamWaitEvent(cudaStream_t, cudaEvent_t)`: No iniciar la ejecución de las tareas del stream hasta que el punto de notificaión del evento se haya alcanzado (en caso de que este esté asociado a otro stream). En el caso que se haya alcanzado el punto de notificación o el evento no este asociado a ningún stream no habrá espera.



(a) La tarea 3 depende de la tarea 1 al ejercutarse secuencialmente por el stream 1. La tarea 2 tiene una dependencia de parámetros con la tarea 1.



(b) La espera por la tarea 1 (sincronización con el stream 1) se realiza justo después de el lanzamiento de la tarea 1.



(c) La espera por la tarea 1 (sincronización con el stream 1) se realiza después de el lanzamiento de la tarea 3. Suponiendo la espera innecesaria de la tarea 2 por la tarea 3.

Figura 4.2: Penalización de tiempo al sincronizar con un stream de dispositivo a destiempo.

Esta funcionalidad de CUDA permite que se establezcan dependencias directas entre streams gestionadas internamente en el dispositivo. Esta abstracción simplifica la complejidad de establecer los puntos de sincronización en la ejecución del hilo, sustituyendo estos puntos por relaciones de dependencia directas entre streams y eventos.

Un stream puede esperar únicamente por un evento por vez, es decir, ante llamadas consecutivas de `cudaStreamWaitEvent`, los eventos dados de introducirán en una cola FIFO que corresponde al stream; este stream irá esperando a un evento por vez (esta cola tiene un tamaño indeterminado no indicado en la documentación oficial de CUDA). En los eventos CUDA, al realizar una llamada `cudaEventRecord` múltiples veces sobre un mismo stream, se sobreescribirá el punto de notificación dentro del evento. Esta sobre-escritura no afecta a los stream que estén esperando por el estado anterior del evento. Sin embargo, dificulta el control de dependencias sobre el stream asociado al Controller, como se describe más adelante.

En esta aproximación se añade a Controllers una estructura de control llamado `CAL_Request`. Esta estructura contiene punteros a un stream y un evento de CUDA. Esta estructura sirve de centinela de una tarea, conociendo el stream de ejecución y su estado.

```
1 typedef struct {
2   void * stream;
3   void * event;
4 } CAL_Request;
```

Figura 4.3: Definición de estructura de control de la librería Controllers.

Las características de los eventos CUDA, tal y como acabamos de mencionar, hacen necesario la creación de estructuras de control independientes por cada lanzamiento de kernel (tarea de computación) y una instancia por cada estructura de datos HitTile asociada al Controller. Esta inicialización de las estructuras supuso una penalización de $\times 5$ veces el tiempo de ejecución respecto a la aproximación anterior, sobre ejecuciones con una gran cuantía de tareas independientes de computación. La penalización de las estructuras de control, que aprovechan las funcionalidades de los streams y eventos CUDA, supuso un cambio en el modelo propuesto.

4.2.3. Aproximación final: política de dependencias interna

Esta sección describe la implementación el modelo propuesto en la sección 3.3. Además, se describen las políticas seguidas para los puntos de sincronización en la dependencia de datos entre las operaciones de comunicación y los lanzamientos de kernels. También se diferencian los estados posibles de una instancia de Controller en la ejecución de un programa con operaciones asíncronas. La implementación activa, en tiempo de ejecución la política del modelo propuesto, mediante la variable de entorno `CAL_CNTRL_ASYNC_MODE`, asignando el valor 1.

Modificaciones sobe estructura HitTile

Un HitTile (en adelante también llamado *tile*), como se describió en la apartado 2.2.1, almacena meta-data de una estructura de datos. La información que esta estructura almacena sobre el Controller corresponde a los siguientes miembros de la estructura:

`refCntrl`, puntero a la instancia de Controller a la que está asociada;

`devData`, puntero a la dirección de memoria de la imagen de la estructura en el dispositivo;

`inCntrlType`, identificador de tipo de estructura que guarda el HitTile (aquellas variables que solo se encuentran en el dispositivo toman el nombre de *internal*);

`transferred`, flag que determina si una estructura ha sido transferida al dispositivo asociado a la instancia de Controller;

`recover`, flag que determina si la información de una estructura ha sido recuperada por la máquina host.

El modelo de sincronización implementado, que evita los problemas de concurrencia entre los hilos involucrados en la ejecución de Controllers, se apoya en los siguientes miembros añadidos al meta-data de la estructura:

`handler`, referencia al stream asociado a la estructura;

`hasPendingMoveTo`, flag que determina si la estructura de datos tiene pendiente una transferencia al dispositivo;

`hasPendingMoveFrom`, flag que determina si la estructura de datos tiene pendiente una transferencia desde el dispositivo;

`isCntrlBlocked`, flag que determina si el hilo de la instancia de Controller está bloqueada por la estructura de datos debido a una comunicación;

`cntrlKernelCount`, contador de kernels que tienen como parámetro de entrada a la estructura de datos.

Implementación de operaciones de comunicación

Este apartado describe las implementaciones de las operaciones de comunicación introducidas en la sección 3.2.

Las transferencias de las estructuras de datos se distinguen según dos criterios:

a) si es necesario o no, reservar/liberar recursos en la máquina host o en el dispositivo;
b) sentido de la transferencia de la comunicación, al/del dispositivo asociado a la instancia del Controller.

El criterio a) distingue las operaciones attach y detach de dos nuevas operaciones de actualización de las estructuras de datos entre la memoria de la máquina host y la del dispositivo: (1) *MoveTo* (transferencia de datos al dispositivo), y (2) *MoveFrom* (transferencia de datos desde el dispositivo). Estas nuevas operaciones implican únicamente transferencias de datos.

Según el criterio b), estas transferencias están implícitas en las dos operaciones del modelo anterior de Controllers: (1) movimiento de datos al dispositivo después de la reserva de recursos (attach), y (2) recuperación de la estructura desde el dispositivo antes de la liberación de memoria en este (detach).

Funciones callback CUDA

Una *función callback*, o *call-after*, es un código ejecutable que se pasa como argumento a otro código, esperando que se llame posteriormente (call back) a este argumento. La ejecución puede ser inmediata

mediante una llamada síncrona, o puede ocurrir después de un tiempo mediante una llamada asíncrona. Existen dos tipos de callback:

1. *callback bloqueantes* (*callback síncronas* o simplemente *callback*, *blocking callbacks*), las cuales son ejecutadas antes del fin de la función principal;
2. *callback diferidas* (*callback asíncronas*, *deferred callbacks*), las cuales son ejecutadas después del retorno de la función principal por un hilo o proceso secundario.

La implementación final del prototipo utiliza callbacks asíncronas de CUDA. Estas funciones son ejecutadas por *el hilo del driver* CUDA asociado al dispositivo. Este hilo se ejecuta en la máquina host y es independiente de la arquitectura de Controllers. Una función de callback debe ser asignada a un stream después de la incorporación de una operación de CUDA. Esta función será ejecutada cuando dicha operación se haya completado. El hilo del driver tiene una cola de callbacks pendientes a la que se irán incluyendo aquellas que estén preparadas para ser ejecutadas; este hilo las evaluará de forma sincrónica mediante política FIFO.

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

Políticas de dependencia de datos entre comunicación y computación

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

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

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

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

La figura 4.4 muestra el diagrama de despliegue con la configuración en tiempo de ejecución de una

aplicación paralela con el prototipo del modelo propuesto. En la parte inferior de la figura 4.4 se encuentra una leyenda que explica las relaciones entre los hilos y las colas. El modelo propuesto contiene tres hilos de ejecución, en la figura 4.4 se encuentran representados estos hilos además de las colas de trabajos existentes entre ellos (vease figura 3.1). Todos y cada uno de estos hilos se ejecutan en la máquina host:

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

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

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

Se ha elegido el uso de semáforos POSIX [2] para los procedimientos de sincronización del modelo propuesto. Los miembros añadidos a la estructura de Controllers para la abstracción sobre dispositivos GPU de NVIDIA (estructura `CALCntrlGPU`) son: `semHost`, semáforo dedicado para el control de sincronización con el hilo principal de la aplicación paralela (parte izquierda de la figura 3.1); `semCntrl`, semáforo dedicado para el control de sincronización con el hilo de la instancia de Controller (parte central de la figura 3.1). Se utilizan dos semáforos distintos ya que el modelo propuesto restringe las sincronizaciones entre los hilos participantes (ver figura 3.1), en las siguientes dos interacciones entre estos hilos:

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

Implementación de políticas de dependencias

Este apartado describe los procedimientos implementados que realiza la librería de Controllers para evitar los problemas de concurrencia descritos en anteriormente. La figura 4.5 muestra el diagrama de estados que pasa la aplicación con el modelo propuesto. Esta figura resume el estado por el que pasan la ejecución de la aplicación, también se indica el estado de bloqueo del hilo principal de la aplicación y el hilo de la instancia del Controller.

La política ideada para resolver dependencia A) (Dependencia entre kernel y comunicación MoveTo) es la siguiente. Esta sincronización involucra los hilos de la instancia de Controller y el del driver de CUDA. Las operaciones envueltas en esta dependencia son: (1) *tarea de comunicación MoveTo* y (2) *lanzamiento de kernel*, ejecutados por el hilo de la instancia de Controller; (3) *callback tras una comunicación MoveTo*, ejecutado por el hilo del driver de CUDA. La figura 4.6 explica las rutinas que se llevan a cabo, respectivamente. La parte izquierda de la figura 4.5 muestra el estado por el que pasa la instancia de Controller ante el lanzamiento de un kernel. El subestado *Normal* define el estado inicial de la instancia, al comenzar a evaluar un lanzamiento de kernel, en el que los hilos de ejecución de la aplicación

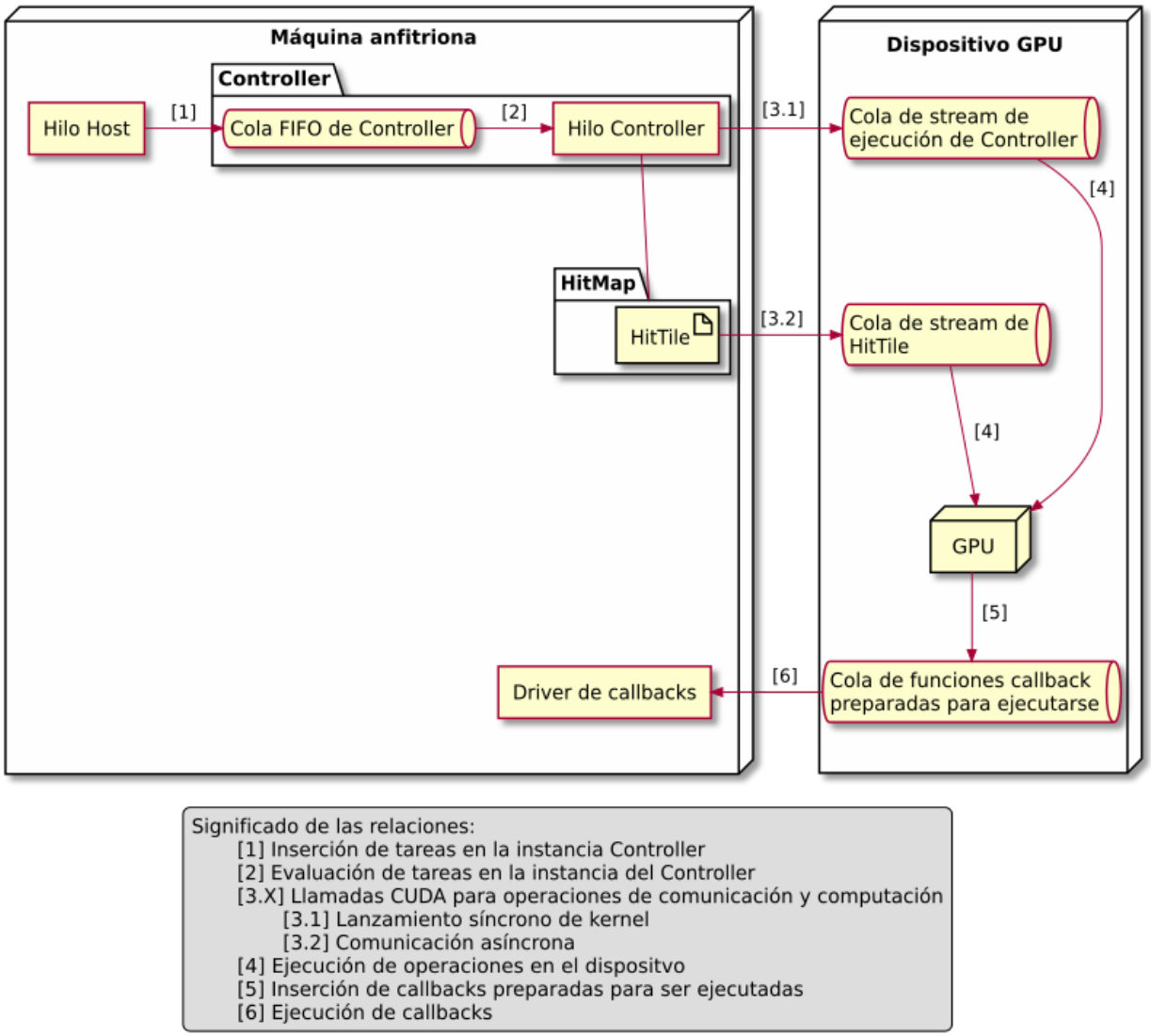


Figura 4.4: Diagrama de despliegue de implementación del modelo propuesto.

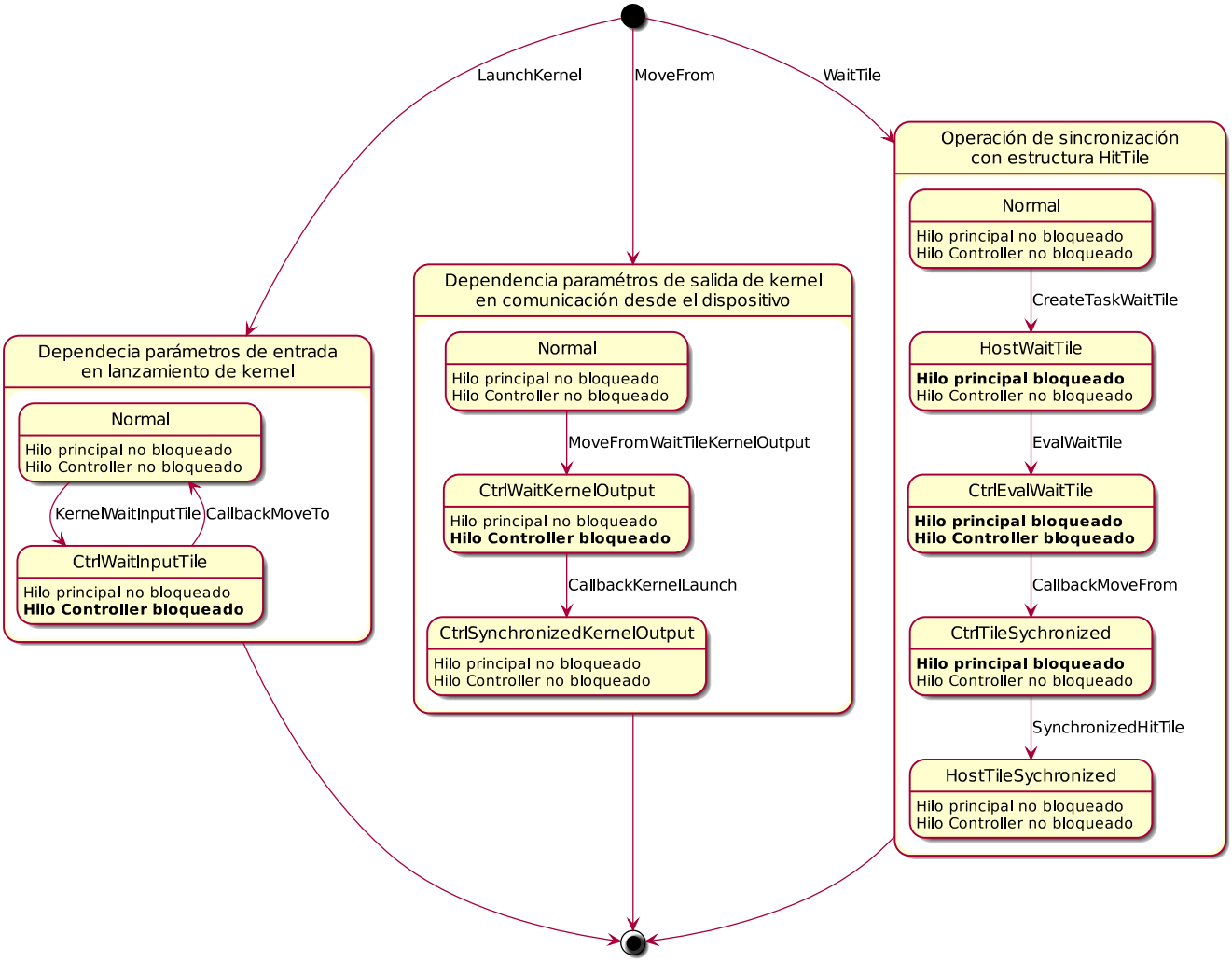


Figura 4.5: Diagrama de estados de instancia de Controller al evaluar una de las tareas involucradas en las políticas definidas.

se mantienen sin bloquear. En caso de que un kernel tuviese que esperar por alguno de los parámetros tiles de entrada, la instacia pasa al subestado de espera por ese tile (*CtrlWaitInputTile*) bloqueando el hilo de la instancia del Controller. Tras la finalización de la comunicación pendiente del tile, el hilo de la instancia se desbloquea volviendo al subestado Normal y siguiendo evaluando los parámetros de entrada.

Procedimiento 4.1: EvaluateMoveTo

```
Entrada:
cntrl: reference of Controller instance;
tile: reference of data structure;
inicio
    tile.hasPendingMoveTo ← true;
    // Ejecución de transferencia
       MoveTo
fin
```

Procedimiento 4.2: EvaluateKernelLaunch

```
Entrada:
cntrl: reference of Controller instance;
kernel: reference of task of kernel launch;
inicio
    para cada tile ∈ kernel.parameters
     hacer
        si tile.role = input entonces
            tile.isCntrlBlocked ← true;
            si tile.hasPendingMoveTo = true
             entonces
                wait(cntrl.semCntrl);
            en otro caso
                tile.isCntrlBlocked ← false;
            fin
        fin
    fin
    // Ejecución del lanzamiento del
       kernel
fin
```

Procedimiento 4.3: CallbackMoveTo

```
Entrada:
tile: reference of data structure;
inicio
    tile.hasPendingMoveTo ← false;
    si tile.isCntrlBlocked = true entonces
        tile.isCntrlBlocked ← false;
        post(tile.refCntrl.semCntrl);
    fin
fin
```

Figura 4.6: Política de dependencia comunicación MoveTo – Kernel.

Para resolver la dependencia B) (Dependencia entre comunicación MoveFrom y kernel). Los hilos involucrados en el procedimientos son los mismos que en el caso anterior: el hilo de la instancia de Controller y el hilo del driver de CUDA. Las operaciones que se ven afectadas por el punto de sincronización son: (1) *lanzamiento de kernel*, (2) y *tarea de comunicación MoveFrom* ejecutados por el hilo de la instancia de

Controller; (3) *callback de lanzamiento de kernel*, ejecutado por el hilo del driver de CUDA. La figura 4.7 explica las rutinas que se llevan a cabo para resolver esta dependencia. La parte central de la figura 4.5 muestra los estados por los que pasa la estancia de Controller ante una transferencia de una estructura de datos desde el dispositivo. El subestado *Normal* define el estado inicial de la instancia, al comenzar a evaluar la transferencia, en el que los hilos de ejecución de la aplicación se mantienen sin bloquear. En caso de que hubiese algún(os) kernel(s) que modique(n) la estructura de datos sin completar, la instancia de Controller pasa al subestado de espera (*CtrlWaitKernelOutput*) bloqueando el hilo de la instancia del Controller. Tras la finalización de la(s) ejecución(es) pendiente(s), el hilo de la instancia se desbloquea pasando al subestado final de *CtrlSynchronizedKernelOutput*.

Finalmente, la estrategia para establecer un punto de sincronización, que bloquee el hilo principal de forma explícita, compromete a los tres hilos de la ejecución del programa paralelo (ver figura 3.1). Las operaciones que se ven afectadas por el punto de sincronización son: (1) *inclusión de tarea de WaitTile*, ejecutada por el hilo principal; (2) *tarea de MoveFrom* y (3) *tarea de WaitTile*, ejecutadas por el hilo de la instancia de Controller; (4) *callback de comunicación MoveFrom*, ejecutado por el hilo del driver de CUDA. El figura 4.8 explica las rutinas que se llevan a cabo, respectivamente. La parte derecha de la figura 4.5 muestra los estados por los que pasa la estancia de Controller ante una transferencia de una estructura de datos desde el dispositivo. El subestado *Normal* define el estado inicial de la instancia, al crear una operación se sincronización con una estructura de datos, en el que los hilos de ejecución de la aplicación se mantienen sin bloquear. Tras la creación e inclusión en la cola de tareas de la instancia del Controller, se pasa al subestado *HostWaitTile*, donde el hilo prinicipal de la aplicación se encuentra bloqueado. Cuando la instancia de Controller extrae la tarea de sincronización de la cola, la evalua y se bloquea esperando a las comunicaciones desde el dispositivo a la máquina host (subestado *CtrlEvalWaitTile*), en caso de que hubiese. Al finalizar la transferencia pendiente, el callback desbloquea el hilo de la instancia del Controller pasando al subestado *CtrlTileSynchronized*. Por último, el hilo de la instancia del Controller desbloquea el hilo princiapl de la aplicación llegando al estado final *HostTileSynchronized*. La estructura de datos habrá terminado su transferencia a la máquina host y habrá consistencia con la imagen de la estructura en el dispositivo.

Todas las operaciones de lectura y escritura de los miembros de una estructura de datos se realizan mediante operaciones atómicas. Es necesario el uso de operaciones atómicas para mantener la concistencia de memoria en las operaciones de lectura y ecritura sobre las estructuras de datos compratidas por los hilos en ejecución.

4.3. Escenarios de ejemplo

Esta sección describe dos escenarios de ejemplo para representar el comportamiento del prototipo del modelo propuesto sobre casos prácticos. Estos casos permite mostrar como el prototipo del modelo propuesto permite aprovechar las ventajas de solapamiento entre comunicaciones y computaciones.

Procedimiento 4.4: EvaluateKernelLaunch

Entrada:
cntrl: reference of Controller instance;
kernel: reference of task of kernel launch;
inicio
  `// Política de dependencia comunicación MoveTo - Kernel`
  para cada $tile \in kernel.parameters$ **hacer**
    si $tile.role = output$ **entonces**
      $tile.cntrlKernelCount \leftarrow tile.cntrlKernelCount + 1$;
    fin
  fin
  `// Ejecución del lanzamiento del kernel`
fin

Procedimiento 4.5: EvaluateMoveFrom

Entrada:
cntrl: reference of Controller instance;
tile: reference of data structure;
inicio
  $tile.isCntrlBlocked \leftarrow true$;
  si $tile.cntrlKernelCount > 0$ **entonces**
    $wait(cntrl.semCntrl)$;
  en otro caso
    $tile.isCntrlBlocked \leftarrow false$;
  fin
  `// Ejecución de transferencia MoveFrom`
fin

Procedimiento 4.6: CallbackKernelLaunch

Entrada:
kernel: reference of task of kernel launch;
inicio
  para cada $tileinkernel.parameters$ **hacer**
    si $tile.role = output$ **entonces**
      $tile.cntrlKernelCount \leftarrow tile.cntrlKernelCount - 1$;
      si $tile.cntrlKernelCount = 0 \wedge tile.isCntrlBlocked = true$ **entonces**
        $tile.isCntrlBlocked \leftarrow false$;
        $post(tile.refCntrl.semCntrl)$;
      fin
    fin
  fin
fin

Figura 4.7: Política de dependencia Kernel – comunicación MoveFrom.

Procedimiento 4.7: CreateTaskWaitTile

Entrada:

cntrl: reference of Controller instance;

tile: reference of data structure;

Datos:

task: initialize task of synchronization with data structure;

inicio

$cntrl.listTask \leftarrow cntrl.listTask \bigcup \{task\}$;

$wait(cntrl.semHost)$;

fin

Procedimiento 4.8: EvaluateMoveFrom

Entrada:

cntrl: reference of Controller instance;

tile: reference of data structure;

inicio

`// Política de dependencia comunicación Kernel - MoveFrom`

$tile.hasPendingMoveFrom \leftarrow true$;

`// Ejecución de transferencia MoveFrom`

fin

Procedimiento 4.9: EvaluateWaitTile

Entrada:

cntrl: reference of Controller instance;

tile: reference of data structure;

inicio

$tile.isCntrlBlocked \leftarrow true$;

si $tile.hasPendingMoveFrom = true$ **entonces**

$wait(cntrl.semCntrl)$;

en otro caso

$tile.isCntrlBlocked \rightarrow false$;

fin

$post(cntrl.semHost)$;

fin

Procedimiento 4.10: CallbackMoveFrom

Entrada:

tile: reference of data structure;

inicio

$tile.hasPendingMoveFrom \leftarrow false$;

si $tile.isCntrlBlocked = true$ **entonces**

$tile.isCntrlBlocked \leftarrow false$;

$post(tile.refCntrl.semCntrl)$;

fin

fin

Figura 4.8: Operación de sincronización con una estructura de datos: Creación de tarea de WaitTile.

4.3.1. Secuencia de multiplicación de matrices

Este caso de ejemplo, representado por el código en la figura 4.9 consiste en una secuencia de multiplicación de matrices.

$$C = A \times B \times D \times E$$

Esta multiplicación se ha planteado de forma iterativa según:

$$C_i = C_{i-1} \times X_i \ ; \ \forall i = 1, 2, 3, \ldots, n$$

```
1  /* Matrix copy movement: Generic kernel code for any type of device */
2  KERNEL(copy, 2, OUT, HitTile_float, dst, IN,  HitTile_float, src){ /* ... */ }
3
4  /* Matrix multiplication: Generic kernel code for any type of device */
5  KERNEL(matMul, 2, OUT, HitTile_float, C, IN,  HitTile_float, A, IN,  HitTile_float, B){ /* ... */ }
6
7  int main() {
8
9     HitTile_float A, B, C, Ci, D, E;
10    // ...
11    CntrlMoveTo(comm, &A);
12    // Copy kernel to initialize Ci: Ci <- A
13    CntrlLaunch(comm, copy, threadsSpace, Ci, A);
14    // Transfer matrix of next iteration to device
15    CntrlMoveTo(comm, &B);
16    // Update kernel of matrix multiplication: C <- Ci x B
17    CntrlLaunch(comm, matMul, threadsSpace, C, Ci, B);
18    // Copy kernel to update Ci for next iteration: Ci <- C
19    CntrlLaunch(comm, copy, threadsSpace, Ci, C);
20
21    CntrlMoveTo(comm, &D);
22    CntrlLaunch(comm, matMul, threadsSpace, C, Ci, D);
23    CntrlMoveTo(comm, &E);
24    CntrlLaunch(comm, copy, threadsSpace, Ci, C);
25    CntrlLaunch(comm, matMul, threadsSpace, C, Ci, E);
26
27
28    // ...
29
30 }
```

Figura 4.9: Código de secuencia de multiplicación de matrices.

Para la iteración inicial en el código de ejemplo (figura 4.9), $C_0 \leftarrow A$ y $X_0 \leftarrow B$. Por cada iteración hay tres operaciones fundamentales: (1) transferencia al dispositivo de la matrix X correspondiente a la iteración; (2) actualización de C_i con el resultado de la matriz C de la iteración previa; (3) multiplicación de matrices para resultado de la iteración actual.

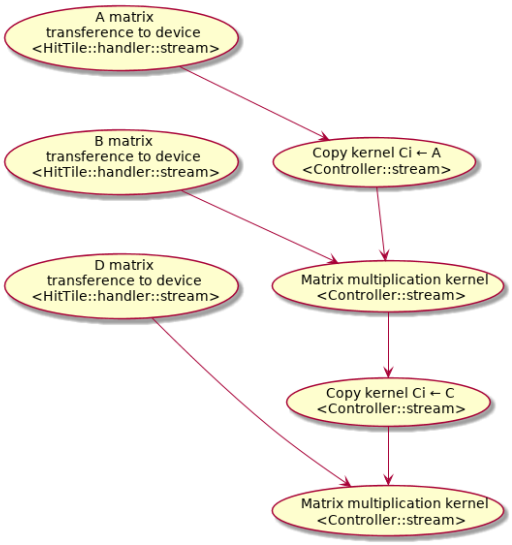


Figura 4.10: Dependencias entre operaciones del ejemplo de secuencia de multiplicación de matrices.

En la línea 11 se mueve A al dispositivo. En la línea 13 se lanza un kernel para inicializar $C_0 \leftarrow A$; como el lanzamiento de este kernel toma A como entrada debe esperar a la finalización de la comunicación. La transferencia de B (línea 15), si podrá solaparse con el kernel de copia previo al no haber intersección entre los parámetros. En cambio, el siguiente kernel de multiplicación de matrices de la línea 17 deberá esperar por esta comunicación. La figura 4.10 muestra las dependencias entre tareas; esta figura muestra en paralelo aquellas operaciones donde puede haber ocasión de solapamiento.

En la figura 4.10 se representa también la iteración de multiplicación con D. La tarea de comunicación de D se solapa con el kernel previo de multiplicación de matrices y la actualización de C_i. El kernel que calcula $C \leftarrow C_i \times D$, se sincroniza con el final de la transferencia de D. Este comportamiento se repite para la siguiente multiplicación que implica a E.

4.3.2. Obtención asíncrona de resultados

Este caso de ejemplo, representado por el código de la figura 4.9 se ha extraído del código utilizado en la sección 2.2.1. Este código consiste en una implementación del método Jacobi-2D para un número determinado de iteraciones.

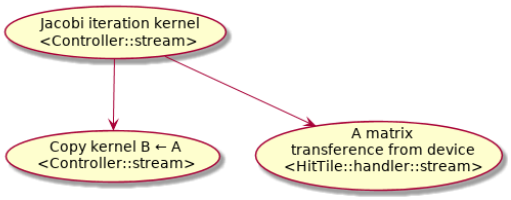


Figura 4.12: Dependencias entre operaciones del ejemplo de Jacobi bi-dimensional.

Para este escenario de prueba nos centramos en el bucle `for` del código de la figura 4.11 entre las líneas 13 y 21. Cada iteración se resume en las siguientes operaciones: (1) cálculo de la iteración de Jacobi de B sobre A; (2) copia de los resultados de la iteración $B \leftarrow A$ para la siguiente iteración; (3) transferencia de A desde el dispositivo a la matriz original; (4) sincronización con la estrucutra A; (5) uso del contenido de A en el hilo principal.

La figura 4.12 representa las dependencias entre las operaciones de comunicación y computación realizadas en el dispositivo. Las tareas de computación son síncronas entre sí, por lo tanto se establece una dependencia implícita del modelo. Sin embargo, la tarea de transferencia desde el dispositivo depende de la salida del kernel de iteración de Jacobi ya que A se encuentra con un rol de salida. Esta dependencia, hace que sea posible el solapamiento del kernel de copia con la transferencia de la estructura de datos. Esta figura no indica la dependencia de la operación de sincronización del hilo principal con A, al tratarse de una operación que no afecta a la ejecución del dispositivo.

```
1  /* Matrix copy movement: Generic kernel code for any type of device */
2  KERNEL(copy, 2, OUT, HitTile_float, dst, IN,  HitTile_float, src){ /* ... */ }
3
4  /* Jacobi iteration: Generic kernel code for any type of device */
5  KERNEL(jacobi, 2, OUT, HitTile_float, A,  IN,  HitTile_float, B ){ /* ... */ }
6
7  int main() {
8
9     HitTile_float A, B, C;
10    // ...
11
12    /* Stage 4: Kernel launchings with a number of logical fine-grain threads */
13    for(int iter=0; iter < NUM_ITERS; iter++)
14    {
15       CntrlLaunch(comm, jacobi, threadsSpace, 2, A, B);
16       CntrlLaunch(comm, copy, threadsSpace, B, A);
17       CntrlMoveFrom(comm, &A);
18
19       CntrlWaitTile(comm, &A);
20       usePartialResultInHost(A);
21    }
22
23    // ...
24 }
```

Figura 4.11: Código de recuperación de resultados de método Jacobi en cada iteración.

4.4. Conclusiones del capítulo

La implementación del prototipo del modelo propuesto permite el solapamiento de tareas, de forma automática y sin problemas de dependencias, basándose en un análisis de dependencias entre estas. Se ha añadido dos tipos de tareas de comunicación que implican únicamente movimiento de datos sobre la implementación anterior de Controllers. Estas dos nuevas operaciones MoveTo y MoveFrom, estaban implícitas en el modelo previo, y son las transferencias sobre las que puede solapar las tareas de computación. La implementación propuesta no supone cambios en el código de aplicaciones ya programadas sobre la implementación anterior de Controllers.

La gestión manual de dependencias entre operaciones de computación y comunicación mediante CUDA se dificulta según la complejidad de la aplicación. Sin embargo, la aproximación final del propotipo abstrae esa complejidad y la reduce a los roles que el usuario da a los parámetros de las operaciones. Esta información se encuentra en la descripción de los prototipos de los kernels involucrados en la aplicación paralela y en el sentido de las transferencia de datos.

Capítulo 5

Experimentación

Este capítulo introduce los siguientes aspectos:

- Los objetivos de la experimentación realizada.
- Los casos de estudio elegidos:
  - Implementación Jacobi bi-dimensional iterativo.
  - Potencia de una matriz.
- Estudio de rendimiento que valide nuestra propuesta.
- Estudio de métricas de software sobre el código generado.
- Los resultados obtenidos de la experimentación y las conclusiones extraídas.

5.1. Objetivos de la experimentación

Esta sección presenta los objetivos de la experimentación del modelo propuesto. Estos objetivos son los siguientes:

1. El solapamiento ofrece mejoras del rendimiento de una aplicación paralela de hasta un 50 % respecto a una versión completamente síncrona.
2. La mejora de rendimiento producido por el solapamiento depende de la carga de trabajo de las tareas de comunicación y computación y de las dependencias existentes entre ellas.
3. La solución propuesta sobre la librería Controllers ofrece una implementación de código con menor complejidad que sobre otras tecnologías existentes.
4. El sobrecoste de la abstracción, introducido por el prototipo implementado y las estructuras de control internas, tiene un efecto mínimo sobre el rendimiento de ejecuciones paralelas comparado con soluciones sobre otras tecnologías existentes. Este sobrecoste tiene un efecto cercano al 10 % sobre el tiempo de código base.

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

Jacobi bi-dimensional: este caso de estudio utiliza el método iterativo de Jacobi para un espacio discreto

bi-dimensional.

Potencia de una matriz: este caso de estudio calcula la potencia de una matriz elevada a un número n dado, $C = A^n$.

Los casos de estudio siguen el esquema del caso de ejemplo explicado en la apartado 4.3.2. En este esquema se produce un posible solapamiento entre la transferencia y el computación de un kernel de copia entre estructuras de datos en el dispositivo. Ambos casos presentan problemas iterativos donde se recupera el resultado de cada iteración.

5.2. Casos de estudio

Esta sección define los casos de estudio introducidos en la sección anterior. Se explica en mayor profundidad cada caso de estudio, describiendo: (1) descripción general del caso de estudio; (2) el algoritmo del problema; (3) la motivación del caso de estudio para el modelo propuesto que se quiere validar.

5.2.1. Caso de estudio: Jacobi bi-dimensional

Este caso de estudio computa la estabilidad puntual de la Ecuación Diferencial Parcial de Poisson (PDE) de la difusión de calor. Utiliza el método iterativo de Jacobi para un espacio discreto bi-dimensional. Este programa stencil de cuatro puntos ejecuta un número de iteraciones fijo. En cada iteración se calcula un nuevo valor por cada celda de la matriz, utilizando la información de los cuatro vecinos superior, inferior, izquierdo y derecho. El resultado después de cada iteración se transfiere a la máquina host. El pseudocódigo 5.2 muestra el algoritmo este caso de estudio.

Este caso de prueba tiene pequeña carga computacional; el kernel de actualización de la iteración únicamente supone el acceso a los vecinos y el cálculo del valor medio. Según la ecuación 3.2 presente en la apartado 3.4.2, al tener mucha diferencia entre los tiempos de comunicación y computación, el ratio entre la suma de los tiempos entre el máximo de ambos será cercano al valor 1. Es decir, no supone mejoras respecto a un modelo puramente síncrono.

Para que haya un balance entre los tiempos de computación y comunicación se debe aumentar la carga de esta primera. Este aumento de carga se consigue con el incremento de la cantidad de datos que se compute, en este caso, el tamaño de la matriz. El incremento de los datos también influye directamente sobre el coste de las comunicaciones, pero los procesos internos de CUDA para el lanzamiento de trabajos producen que la computación se vea más afectada.

Pseudocódigo 5.1: Caso de estudio Jacobi bi-dimensional.

```
Función Copy
    Entrada: B: target matrix, A: matrix copy
    Datos: x, y: column and row indexes respectively
    inicio
        B[y][x] ← A[y][x];
    fin
fin
Función JacobiIteration
    Entrada: A: target matrix, B: matrix copy
    Datos: x, y: column and row indexes respectively
    inicio
        avg ← (B[y − 1][x] + B[y][x − 1] + B[y][x + 1] + B[y + 1][x])/4;
        A[y][x] ← avg;
    fin
fin
Función main
    Datos: n: number of iterations, A: original matrix, B: matrix copy
    inicio
        para i ← 1 a n hacer
            Copy(B, A);
            JacobiIteration(A, B);
            MoveFrom(A);
        fin
    fin
fin
```

5.2.2. Caso de estudio: Potencia de una matriz

Este caso de estudio calcula $C = A^n$. Se elevar una matriz cuadrada A por una potencia n. Hemos escogido una implementación basada en un bucle que computa la expresión recursiva:

$$\begin{aligned} C_1 &= A \\ C_i &= C_{i-1} \times A;\ k = 2, \ldots, n \end{aligned} \tag{5.1}$$

El resultado después de cada iteración se transfiere a la máquina host.

Pseudocódigo 5.2: Caso de estudio de potencia de una matriz.

```
Función main
    Datos: n: number of iterations, A: original matrix, C: result matrix, C_i: temporal result
           matrix
    inicio
        Copy(C_i, A);
        para i ← 2 a n hacer
            MatrixMultiplication(C_i, A, C);
            Copy(C_i, C);
            MoveFrom(C);
        fin
    fin
fin
```

Este caso de estudio presenta una gran carga computacional. El algoritmo convencional de multiplicación de matrices tiene un orden $\mathcal{O}(n^3)$ siendo n el tamaño de la matriz. La implementación de este caso de estudio utiliza un algoritmo de divide y vencerás mediante el uso de memoria compartida y tiene una complejidad teórica de $\mathcal{O}(n^3/\log_2 n)$. El kernel que multiplica ha sido escogido del ejemplo optimizado contenido en los ejemplos de CUDA ofrecidos por NVIDIA. Esta implementación utiliza memoria compartida entre hilos en el dispositivo y desenrollado de bucles para obtener mejoras en el uso de los recursos de la GPU.

El alto coste computacional del caso de estudio hace que se obtengan mejores resultados teóricos según la ecuación 3.2 en la apartado 3.4.2. El alto coste de computación y comunicación hace que el solapamiento entre operaciones produzca mejoras notables ante tamaños de entrada más pequeños que en el caso de estudio anterior.

5.3. Estudio de rendimiento

Esta sección expone el estudio de rendimiento realizado que valida el modelo propuesto. Se describe la plataforma y la metodología realizada en la experimentación. Por último, se muestran los resultados y las conclusiones extraídas de la experimentación.

5.3.1. Plataforma

La experimentación ha sido realizada sobre una máquina con cuatro aceleradoras GPU de NVIDIA. Esta máquina tiene las siguientes características:

- Procesador: Intel Xeon E5-2690 v3 @1.9 GHz
- Memoria RAM principal: 64 GB GDDR3
- Cuatro aceleradoras GPU con las siguientes especificaciones:
  - Modelo: NVIDIA's GTX Titan BLACK
  - Arquitectura: Kepler
  - Núcleos: 2880 cores con frecuencia 980 MHz
  - Memoria de vídeo: 6 GB GDDR5
  - Versión de CUDA: V9.0.102

5.3.2. Metodología

Se han desarrollado versiones síncronas y asíncronas de soluciones con la tecnología CUDA para los problemas escogidos. Nos referimos a estas soluciones como las versiones base para la comparación de rendimiento. El código de Controllers tiene una única implementación, y la política de transferencias asíncronas se activa en tiempo de ejecución. El estudio de rendimiento comprara los tiempos totales entre:

1) implementaciones programadas con la librería de CUDA y otra utilizando la librería de Controllers;
2) versiones síncorna y asíncrona de Controllers, utilizando el modelo propuesto.

En ambos casos de estudio, los tiempos tomados en las distintas versiones acotan el bucle iterativo. Las ejecuciones de las soluciones desarrolladas se ejecutan variando el tamaño de las matrices del problema y el número de iteraciones del bucle principal del caso de estudio. Por cada combinación de número de iteraciones y tamaño de entrada, se han repetido las ejecuciones 10 veces y se ha obtenido la media del tiempo de las ejecuciones. La tabla 5.1 muestra estos valores.

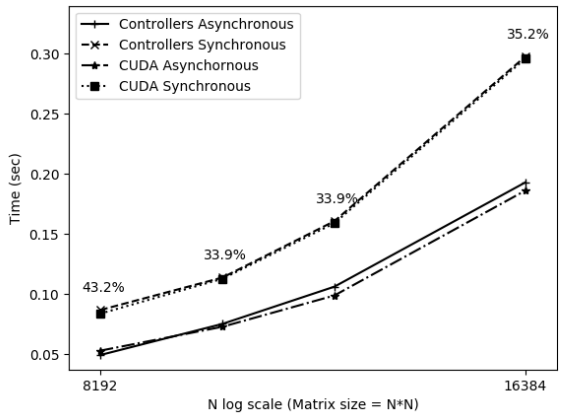
Caso de estudio	Números de iteraciones	Tamaños de entradas
Jacobi bi-dimensional	2, 10, 100, 500	8192, 10000, 12000, 16384
Potencia de una matriz	5, 7, 10, 15	512, 1024, 2048, 4096

Tabla 5.1: Número de iteraciones y tamaños de entradas elegidos para la experimentación de rendimiento en cada caso de estudio.

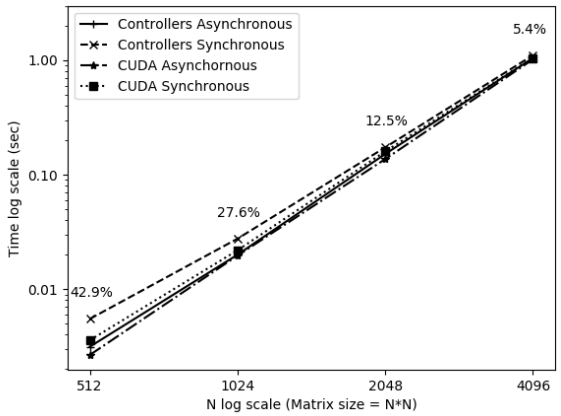
5.3.3. Resultados

Las gráficas contenidas en la figura 5.1 muestra los resultados de rendimiento obtenidos para los casos de estudio. El eje ordenadas representa el tiempo total de ejecución (incluyendo los tiempos computación y comunicación); en el caso de estudio de la potencia de una matriz, el eje y está en escala logarítmica decimal. Los diferentes tamaños de las matrices se representan en el eje de abcisas con escala logarítmica

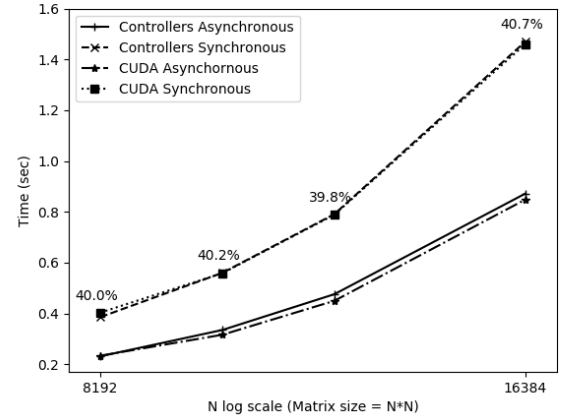
en base 2. En las gráficas, se añaden etiquetas por cada tamaño dado el porcentaje de mejora entre las ejecuciones de la version de Controllers (*Controller Asynchronous* vs *Controller Synchronous*).



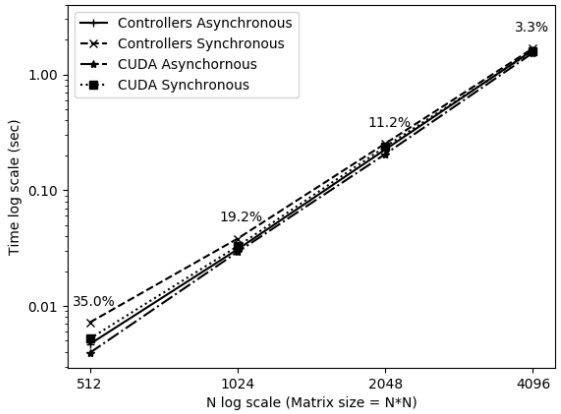
(a) Gráfica comparativade rendimiento – Jacobi bi-dimensional con 2 iteraciones



(b) Gráfica comparativade rendimiento – Elevar una matriz a la décima potencia: $C = A^5$



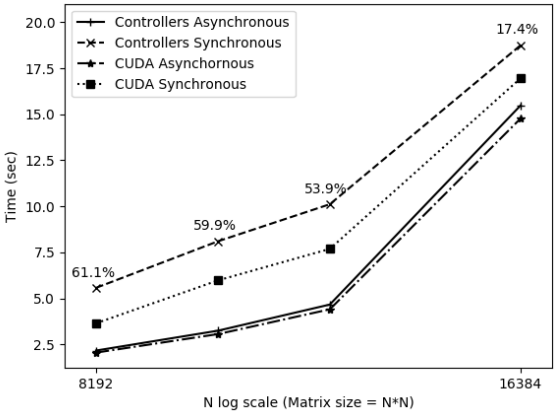
(c) Gráfica comparativade rendimiento – Jacobi bi-dimensional con 10 iteraciones



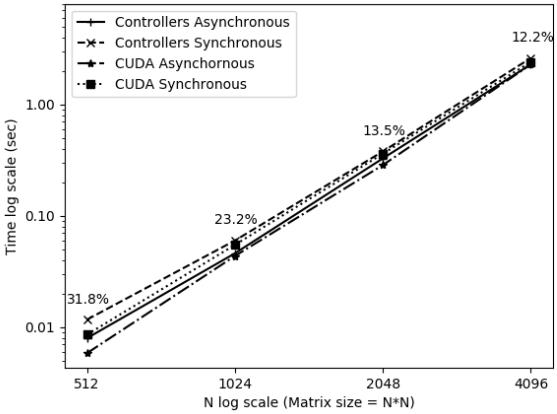
(d) Gráfica comparativade rendimiento – Elevar una matriz a la décima potencia: $C = A^7$

En la gráficas de la figura 5.1, se observan las mejoras de tiempo debidas al solapamiento de tareas de comunicación y computación. En el caso de estudio de Jacobi bi-dimensional, el porcentaje de rendimiento entre los modelos síncrono y asíncrono de Controllers se encuentran entre 17,4 % (véase figura 5.1e) y 61,1 % (véase figura 5.1e). En el caso de estudio de la potencia de una matriz, los porcentajes de mejora están entre 0,8 % (véase figura 5.1h) y 42,9 % (véase figura 5.1b). Algunos porcentajes de mejora, como los mostrados en la figura 5.1e (caso de estudio de Jacobi bi-dimensional), se encuentra por encima del límite teórico del 50 % calculado en la sección 3.4. Este resultado se debe a que el modelo utiliza dos mejoras complementarias sobre la versión síncrona: (1) solapamiento de operaciones de comunicación y computación; y (2) uso de memoria pinned.

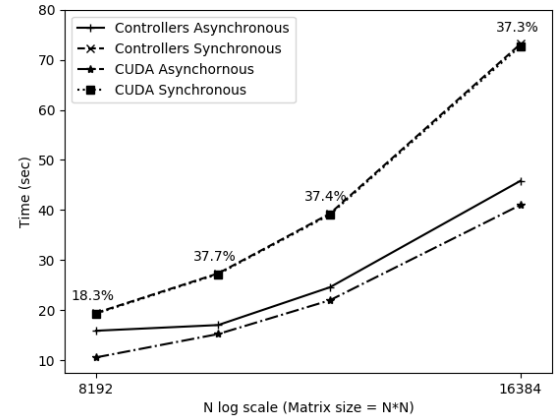
Ambos casos de estudio, el rendimiento de las soluciones síncronas y las soluciones asíncronas muestra tendencias de crecimiento similares entre sí según se incrementa el tamaño de entrada. Las versiones de Controllers muestran una pequeña penalización temporal, respecto a las soluciones en CUDA, debidas al coste extra de las llamadas internas de la librería. Las tablas 5.2a y 5.2b muestran los valores de esta



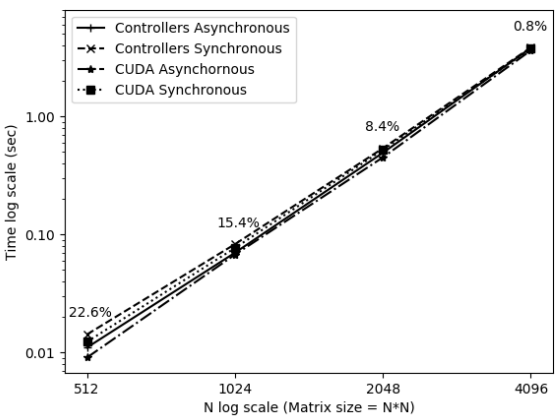
(e) Gráfica comparativade rendimiento – Jacobi bi-dimensional con 100 iteraciones



(f) Gráfica comparativade rendimiento – Elevar una matriz a la décima potencia: $C = A^{10}$



(g) Gráfica comparativade rendimiento – Jacobi bi-dimensional con 500 iteraciones



(h) Gráfica comparativade rendimiento – Elevar una matriz a la décima potencia: $C = A^{15}$

Figura 5.1: Gráficas comparativas de rendimiento entre las versiones implementadas en los casos de estudio Jacobi bi-dimensional y la potencia de una matriz.

penalización en las pruebas realizadas por cada caso de estudio. Destacan los casos de raros producidos por situaciones puntuales, tales como los sobrecostes negativos (mejora de rendimiento) con un tamaño de matriz de 8192 en el caso de Jacobi bi-dimensional, o el sobrecoste de 50,21 % con este mismo tamaño de entrada y un número de 500 iteraciones.

Iteraciones	Tamaño $N \times N$	Sobrecoste(%)
2	8192	−6,85
2	10000	3,26
2	12000	7,43
2	16384	3,65
10	8192	−1,07
10	10000	5,93
10	12000	5,78
10	16384	2,77
100	8192	5,42
100	10000	6,05
100	12000	5,87
100	16384	4,80
500	8192	50,21
500	10000	11,75
500	12000	11,79
500	16384	11,73

(a) Tabla de sobrecoste temporal en el caso de estudio de Jacobi bi-dimensional

Potencia n	Tamaño $N \times N$	Sobrecoste(%)
5	512	18,05
5	1024	1,75
5	2048	10,75
5	4096	2,81
7	512	18,94
7	1024	5,26
7	2048	9,58
7	4096	5,70
10	512	36,41
10	1024	5,68
10	2048	14,67
10	4096	0,22
15	512	21,56
15	1024	3,71
15	2048	9,42
15	4096	4,99

(b) Tabla de sobrecoste temporal en el caso de estudio de potencia de una matriz

Caso de estudio	Media	Mediana	Desviación estándar
Jacobi 2D	8,03 %	5,83 %	12,19
Potencia de matriz	10,59 %	7,56 %	9,43

(c) Tabla de estadísticos del sobrecoste temporal del uso de Contoller

Tabla 5.2: Tablas de sobre-costes temporales de la versión asíncrona con Controllers sobre la versión asíncrona CUDA.

La tabla 5.2c muestra los estadísticos del sobrecoste del uso de la librería Controllers sobre la solución asíncrona base CUDA. En el caso de estudio de Jacobi bi-dimensional los estadísticos de media y mediana del sobrecoste de los experimentos realizados se encuentra por debajo del 10 %. En el caso de la potencia de una matriz la media se encuentra 0,59 % por encima de este valor; en cambio la mediana se encuentra por debajo del límite objetivo que propusimos. Estos valores muestran como la penalización de la abstracción de Controllers no supone un coste de tiempo alto.

5.4. Estudio de métricas de calidad de software

Esta sección expone el estudio de métricas de calidad de software realizado que valida el modelo propuesto. Se describe las métricas usadas en la experimentación. Por último, se muestran los resultados y las conclusiones extraídas de la comparación de las métricas entre los códigos desarrollados para la experimentación de rendimiento.

Este estudio analiza las diferencia en el esfuerzo que realiza el desarrollador entre el código basado en Controllers y la implementación base asíncrona usando CUDA. Hemos medido tres métricas clásicas de calidad de software: (1) número de líneas de código; (2) número de tokens; (3) complejidad ciclomática de McCabe [16]. Las primeras dos métricas miden el volumen de código que un desarrollador debe programar. La tercera mide la divergencia del código y la posibles dificultades potenciales que este puede suponer para su codificación, prueba y depuración.

Las métricas incluyen el código de la definición de los kernels, la caracterización de los prototipos de los kernels, la gestión de las estructuras de datos y el bucle iterativo de los casos de estudio. La definición de los kernels contienen un volumen grande de código que será igual para todos los casos analizados. La tabla 5.3 muestra los valores de las métricas por implementación. En la tabla 5.3 se ve como la implementación de Controllers tiene menos volumen de código. La complejidad ciclomática también se ve reducida en comparación con la versión base CUDA, esto indica que que será necesario un esfuerzo menor de desarrollo.

Caso de estudio	Implementación	NLOC	CCN	token
Jacobi 2D	CUDA	114	5	849
Jacobi 2D	Controller	91	4	711
Potencia de matriz	CUDA	146	5	1053
Potencia de matriz	Controller	115	4	885

Tabla 5.3: Comparativa de métricas de software.

5.5. Conclusiones del capítulo

El solapamiento de tareas de comunicación y computación supone mejoras de rendimiento de hasta un 50 % respecto a la versión puramente síncrona siempre que no haya dependencias entre ellas. Además, se han obtenido porcentajes de mejora superiores al límite teórico debido al uso de memoria pinned que permite transferencias DMA(acceso directo a memoria), es decir, sin intervención de la CPU. El uso de la librería Controllers en comparación como alternativa a una versión base con CUDA tiene un pequeño sobrecoste, por debajo del 10 % de media, debido a las funcionalidades internas de esta primera. El código que utiliza la librería Controllers supone un menor esfuerzo de desarrollo que una implementación manual en CUDA que permita el solapamiento de tareas.

Capítulo 6

Conclusiones

Este capítulo introduce los siguientes aspectos:

- Definición de las conclusiones extraídas del trabajo
- Propuesta de futuras líneas de trabajo
- Exposición de una valoración personal sobre el desarrollo del trabajo

6.1. Objetivos cumplidos

Este trabajo ha formado parte de un proyecto de investigación dirigido por el Grupo Trasgo. Este trabajo presenta una propuesta de modelo para solapar tareas de comunicación y computación en dispositivos GPU. Se ha establecido una política de dependencias de datos para que la ejecución solapada de ambos tipos de tareas no produzca inconsistencias. El usuario no necesita hacer uso a nuevas funciones sobre la versión anterior de la librería Controllers, para explotar las funcionalidades desarrolladas en este trabajo. El uso del modelo propuesto ha obtenido un porcentaje hasta del 50 % de mejora frente a la versión secuencial de Controllers, en la experimentación realizada, mediante el solapamiento de tareas. Además, se ha demostrado que el modelo reduce el esfuerzo de desarrollo que tiene que realizar un usuario. Los objetivos específicos conseguidos son los siguientes:

1. Se han estudiado los trabajos existentes relacionados con la propuesta de este trabajo.
2. Se ha estudiado las librerías de Hitmap y Controllers, y se ha entendido el modelo previo Controllers.
3. Se ha diseñado un modelo automático y transparente para el usuario que solapa operaciones de comunicación y computación.
4. Se ha logrado un estudio y evaluación (test de rendimiento) que valida el prototipo del modelo propuesto.
5. Se han diseñado dos casos de estudio que prueban el solapamiento de operaciones.

El resultado del proyecto a resultado en una publicación que ha sido aceptada y será presentada en las **XXIX Jornadas de Paralelismo** (JP2018) que se celebrará en Teruel el próximo mes de septiembre.

6.2. Trabajo futuro

Las soluciones, obtenidas de este trabajo fin de grado se están utilizando en la librería Controllers para un modelo de programación paralela simple y portable entre diferentes plataformas de ejecución.

Este proyecto queda abierto para futuras ampliaciones y modificaciones. Se pretende añadir más funcionalidades a la biblioteca para permitir explotar otro tipo de aceleradoras hardware y entornos heterogéneos. Además, queremos ampliar el estudio a otras políticas más sofisticadas de gestión de colas, que tengan en cuenta las limitaciones del entorno y las especificaciones del hardware utilizado para la ejecución de las aplicaciones.

Otro trabajo que pretendemos abordar es la integración de una gestión automática de memoria. Este trabajo futuro permitiría evitar la ocupación total de la memoria de un dispositivo en la ejecución de una aplicación. El modelo tiene dos posibles líneas de investigación: a) la reordenación de las tareas para evitar la ocupación total de la memoria disponible; b) la división, de forma automática, de la carga de trabajo de las tareas en subtareas más pequeñas mediante técnicas de *tiling* y las funcionalidades de la librería Hitmap.

6.3. Valoración personal

Como todo trabajo fin de grado, presenta un reto a nivel personal para todo alumno que lo realice. Es una fase de aprendizaje donde se acerca el trabajo que se realiza al mundo real, al mundo donde en un futuro próximo formaremos parte de él. Este trabajo de investigación ha tenido una estructura completamente distinta al desarrollo de software convencional, como el que se plantea en las asignaturas del grado. Las etapas en comparación con un desarrollo son más lentas y metodologías ágiles de trabajo pueden no ser convenientes. La búsqueda bibliográfica para encontrar otros trabajos relacionados requiere bastante tiempo para establecer unas bases de la investigación. Este proyecto ha seguido la base del estudio científico con las fases de: (1) observación y estudio, (2) inducción, (3) planteamiento de hipótesis, (4) probar hipótesis con experimentación, (5) demostración (antítesis) de la hipótesis, y (6) teoría científica. Es por ello que la investigación tiene un proceso más largo y más cercano a la ingeniería tradicional y al método de trabajo en cascada. Este proyecto ha visto muchos cambios en la hipótesis, como se ha visto en la evolución de las aproximaciones, y la experimentación ha llevado tiempo al tener que probarse sobre problemas con cargas de trabajo grandes.

Como estudiante, este trabajo ha servido para afianzar los conocimientos adquiridos en algunas asignaturas del grado. Conocimientos de computación paralela adquiridas en la asignatura del mismo nombre, dada por miembros del Grupo Trasgo, o conocimientos de concurrencia obtenidos en la asignatura de Fundamentos de Sistemas Operativos cursada en segundo curso del grado. También me ha motivado para seguir un camino distinto al que me planteé en un principio. Este trabajo ha hecho decidirme a continuar mi carrera profesional con el estudio del master y doctorado, especializándome en una disciplina con tanto futuro como la computación paralela.

Bibliografía

[1] Consistencia Secuencial - Administración de Memoria Compartida Distribuida. Último acceso: 2018-06-13.

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

[3] How to Overlap Data Transfers in CUDA C/C++, December 2012. Último acceso: 2018-06-18.

[4] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. Último acceso: 2018-06-11.

[5] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. A Dynamic Self-scheduling Scheme for Heterogeneous Multiprocessor Architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, January 2013. Último acceso: 2018-06-06.

[6] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998. Último acceso: 2018-05-07.

[7] Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos. MPI-aware Compiler Optimizations for Improving Communication-computation Overlap. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 316–325, New York, NY, USA, 2009. ACM. Último acceso: 2018-05-03.

[8] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Commun. ACM*, 8(9):569–, September 1965. Último acceso: 2018-06-11.

[9] J. Dongarra, H. Meuer, and E. Strohmaier. Top500 Supercomputer Sites. Technical report, University of Tennessee, Knoxville, TN, USA, 1998.

[10] Ian Buck Tim Foley, Daniel Horn Jeremy Sugerman, Kayvon Fatahalian Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. page 10.

[11] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[12] Arturo Gonzalez-Escribano, Yuri Torres, Javier Fresno, and Diego R. Llanos. An Extensible System for Multilevel Automatic Data Partition and Mapping. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1145–1154, May 2014. Último acceso: 2018-04-30.

[13] Tobias Gysi, Jeremia Bar, and Torsten Hoefler. dCUDA: Hardware Supported Overlap of Computation and Communication. pages 609–620. IEEE, November 2016. Último acceso: 2018-06-05.

[14] Rafia Inam. An Introd uc tion to GPGPU Prog ra m m ing - CUDA Arc hitec ture. page 21.

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

[16] T. J. McCabe. A Complexity Measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976. Último acceso: 2018-05-30.

[17] Ana Moreton–Fernandez, Hector Ortega–Arranz, and Arturo Gonzalez–Escribano. Controllers: An abstraction to ease the use of hardware accelerators. *The International Journal of High Performance Computing Applications*, page 109434201770296, May 2017. Último acceso: 2018-04-30.

[18] Girija J. Narlikar and Guy E. Blelloch. Pthreads for dynamic and irregular parallelism. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–16, Washington, DC, USA, 1998. IEEE Computer Society.

[19] Borja Pérez, José Luis Bosque, and Ramón Beivide. Simplifying Programming and Load Balancing of Data Parallel Applications on Heterogeneous Systems. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, GPGPU '16, pages 42–51, New York, NY, USA, 2016. ACM. Último acceso: 2018-05-03.

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

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

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

[23] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *IEEE Des. Test*, 12(3):66–73, May 2010. Último acceso: 2018-05-07.

[24] Antonio Vilches, Rafael Asenjo, Angeles Navarro, Francisco Corbera, Rub'en Gran, and María Garzarán. Adaptive Partitioning for Irregular Applications on Heterogeneous CPU-GPU Chips. *Procedia Computer Science*, 51:140–149, January 2015. Último acceso: 2018-06-05.

[25] M. Wolfe. More Iteration Space Tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, pages 655–664, New York, NY, USA, 1989. ACM. Último acceso: 2018-06-13.

[26] Chao-Tung Yang and Shun-Chyi Chang. A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters. In G. Goos, J. Hartmanis, J. van Leeuwen, Peter M. A. Sloot, David Abramson,

Alexander V. Bogdanov, Yuriy E. Gorbachev, Jack J. Dongarra, and Albert Y. Zomaya, editors, *Computational Science — ICCS 2003*, volume 2660, pages 1079–1088. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. Último acceso: 2018-06-13.

Apéndices

Apéndice A

Contenidos del CD-ROM

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 .csv con los resultados de las pruebas realizadas.