Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
(Mención en Ingeniería de Software)

Modelo de ejecución asíncrono para sistemas heterogéneos con OpenCL

Alumno:
D. Victor Lara Mongil

Tutores:
Dr. Arturo González Escribano
Dr. Jorge Fernández Fabeiro

RESUMEN

La computación de alto rendimiento HPC está en auge en multitud de campos de la ciencia como la simulación experimental, la meteorología, la inteligencia artificial, el Big Data o el minado de cripto-monedas. Utilizan algoritmos paralelos que se ejecutan en los grandes centros de computación, como los presentes en la lista TOP500 que contiene el top 500 de los mayores supercomputadores del mundo, actualizada mensualmente.

Estos centros de computación son enormemente heterogéneos, con multitud de arquitecturas, dispositivos, co-procesadores, plataformas y tecnologías diferentes. Conseguir aprovechar las capacidades de esos sistemas heterogéneos requiere de conocimiento experto en cada una de esas arquitecturas, además de conocer la configuración exacta del sistema sobre el que se ejecuta. De este modo, el auge de los dispositivos GPGPU (General-Purpose Graphics Processing Units) y sus cada vez mayores capacidades de cómputo han hecho que muchos de los grandes centros de computación utilicen estos dispositivos como principal fuente de potencia de cálculo.

OpenCL es un estándar que provee de una arquitectura y unos mecanismos de programación comunes a diversos tipos de dispositivos, entre ellos GPUs. OpenCL proporciona una portabilidad de código, pero no tiene por qué proveer, en todos los casos, portabilidad de rendimiento, como en el caso de las GPUs de NVIDIA.

Controllers es una librería creada por el Grupo de Investigación Trasgo del Departamento de Informática de la Universidad de Valladolid que ofrece portabilidad de código y portabilidad de rendimiento sobre diferentes tipos de dispositivos (multi-core CPUs, co-procesadores Intel Xeon-Phi, GPUs de NVIDIA). Sin embargo, para sistemas heterogéneos basados en GPUs tiene una portabilidad muy limitada, al no admitir más arquitecturas que las GPUs de NVIDIA.

Este trabajo implementa soporte para OpenCL a la librería de *Controllers*, mejorando su portabilidad de código y su portabilidad de rendimiento, para GPUs de distintas arquitecturas, equiparándose al rendimiento de las tecnologías nativas de las mismas. Además revisa el modelo de ejecución asíncrono que permite el solapamiento de tareas en este tipo de dispositivos, y lo actualiza con una nueva propuesta que mejora el solapamiento conseguido por la librería, aumentando su rendimiento en todo tipo de GPUs.

TABLA DE CONTENIDOS

1

INTRODUCCIÓN

Este capítulo contiene:

- Contexto, motivación y planteamiento del problema.
- Objetivos de la solución propuesta.
- Estructura de este trabajo.

1.1 CONTEXTO, MOTIVACIÓN Y PLANTEAMIENTO DEL PROBLEMA

1.1.1 *Contexto*

La HPC (High Performance Computing o Computación de Alto Rendimiento) surge de la evolución de las computadoras que tienen cada vez una mayor potencia de cálculo. La HPC utiliza técnicas como la computación paralela, la computación concurrente o la computación distribuida para conseguir mayor rendimiento en programas que resuelven problemas de alto coste computacional, ejecutados en super-computadoras. Se utiliza en una amplia variedad de campos científicos y tecnológicos. Algunos ejemplos son la simulación, la meteorología, la inteligencia artificial, el Big Data [6] o las predicciones meteorológicas. La HPC reduce el tiempo necesario de cómputo de este tipo de problemas, en algunos casos de forma drástica.

Cada dispositivo de cómputo (co-procesadores, CPUs, GPUs, etc) tiene su propia arquitectura hardware, con unas capacidades de cómputo concretas, un conjunto de instrucciones propio de su arquitectura y unos mecanismos o tecnologías de programación asociadas (MPI, OpenMP, CUDA, OpenCL, ...). Un vistazo al TOP500 [7], la lista con los 500 mayores supercomputadores del planeta, muestra que actualmente los grandes sistemas de computación son heterogéneos, con múltiples arquitecturas y plataformas. La necesidad de explotar conjuntamente los dispositivos y plataformas de esos sistemas dio lugar a la computación heterogénea [3].

Sin embargo, esta heterogeneidad complica especialmente el desarrollo de aplicaciones con esas plataformas y tecnologías, ya que para exprimir las capacidades de estos sistemas se requiere

tanto el dominio de los conocimientos específicos acerca de cada plataforma y tecnología implicadas como de la configuración exacta de cada sistema sobre el que se ejecute la aplicación. Entre los diferentes intentos de solucionar este problema cobra especial relevancia el estándar OpenCL [1], que proporciona una interfaz común de programación para aplicaciones paralelas. Ofrece una generalización de la arquitectura de los sistemas heterogéneos (host + co-procesadores) y una abstracción de múltiples mecanismos de programación disponibles en diferentes tipos de dispositivos. De esta forma, OpenCL ofrece portabilidad de código para sistemas heterogéneos.

1.1.2 *Motivación*

Para conseguir aprovechar en profundidad los recursos hardware en sistemas heterogéneos, y conseguir un rendimiento lo más cercano posible al que producen las tecnologías nativas de cada plataforma en sistemas heterogéneos, surge la necesidad de una abstracción de las mismas. Una interfaz de programación que proporcione las funcionalidades comunes a las tecnologías nativas subyacentes y que las implemente de acuerdo a sus modelos nativos, muy diferentes entre sí.

OpenCL [1] proporciona una solución al problema de la portabilidad de código, ya que el mismo código OpenCL puede ejecutarse sobre diversas plataformas. Sin embargo, no proporciona portabilidad de rendimiento de forma automática, es decir, no tiene por qué conseguir directamente el mejor rendimiento posible en una plataforma dada.

La librería *Controllers* es uno de los principales proyectos de investigación del Grupo de Investigación Trasgo del Departamento de Informática de la Universidad de Valladolid. Esta librería plantea otra aproximación a este problema, soportando CPUs multinúcleo, co-procesadores Intel XeonPhi [9] como GPUs NVIDIA (estas últimas haciendo uso de su tecnología nativa CUDA) mediante la abstracción de una serie de funcionalidades comunes a las aplicaciones de paralelismo, y automatizando y ocultando al usuario estrategias de mejora de rendimiento complejas propias de cada arquitectura. Así, uno de los problemas que estas estrategias deben abordar es la gestión eficiente de tareas de cómputo y transferencia de datos, de modo que se maximize el rendimiento obtenido. *Controllers* es capaz de realizar un análisis en tiempo de ejecución de las dependencias entre datos para ejecutar esas transferencias de forma eficiente.

Conseguir portabilidad de código y de rendimiento permitiría escribir aplicaciones paralelas portables y eficientes para los sistemas heterogéneos en auge, como los grandes sistemas de supercomputación presentes en la lista TOP500 [7].

1.1.3 *Planteamiento del problema*

Los sistemas de supercomputación actuales son intrínsecamente heterogéneos, con múltiples plataformas y dispositivos. Los co-procesadores de tipo GPU son uno de los tipos de dispositivos en auge y los sistemas de super computación más importantes (TOP500) muestran claramente una tendencia hacia los mismos.

La librería *Controllers* ofrece cierta portabilidad de rendimiento al soportar el solapamiento de tareas, acercándose al rendimiento de las tecnologías nativas sobre aquellas plataformas que soporta. Sin embargo, actualmente este soporte se limita a co-procesadores GPU de NVIDIA [10].

1.2 OBJETIVOS DE LA SOLUCIÓN PROPUESTA

1.2.1 *Propuesta*

La propuesta de este trabajo es investigar las posibilidades de añadir soporte para OpenCL a la librería *Controllers*, con el objetivo de poder programar con ella cualquier sistema de computación heterogénea. Supone una mejora drástica en portabilidad del código escrito con la librería, manteniendo su portabilidad de rendimiento.

Además este trabajo analiza la solución que provee el modelo asíncrono de ejecución de Controladores y la gestión de las dependencias entre las transferencias de datos y la ejecución de computación y propone, implementa y prueba una nueva solución, implementada sobre OpenCL

Se mantienen las características y funcionalidades de la librería, haciendo transparente al usuario el uso del modelo de ejecución asíncrono y su gestión de dependencias. Revisar el modelo de dependencias nos permite acercarnos todavía más al rendimiento nativo de las tecnologías soportadas.

1.2.2 *Objetivos*

Los objetivos específicos de este trabajo, definidos siguiendo una metodología de investigación experimental, son:

- Analizar y entender un modelo previo de comunicación asíncrona para co-procesadores (GPU's).

- Proponer un modelo alternativo para computación heterogénea, sin importar el tipo de dispositivo.
- Implementar el modelo propuesto utilizando el estándar de computación heterogénea OpenCL.
- Validación experimental del modelo propuesto.
- Estudio experimental del impacto del modelo en aplicaciones reales.

1.3 ESTRUCTURA DE ESTE TRABAJO

El trabajo presenta la siguiente estructura:

- En el capítulo 2 se detallan los conceptos, términos, tecnologías y plataformas utilizadas en este trabajo, incluida la librería de Controladores y su funcionamiento.
- En el capítulo 3 se discute un nuevo modelo para Controladores y su gestión de dependencias.
- En el capítulo 4 se detalla la implementación de dicho modelo sobre el estándar OpenCL.
- En el capítulo 5 se valida el modelo de forma experimental en base a una batería de pruebas detallada.
- En el capítulo 6 se realiza un estudio experimental del nuevo modelo sobre aplicaciones paralelas reales.
- En el capítulo 7 se plasman las conclusiones y se proponen ideas de trabajo futuro.

2

CONCEPTOS PREVIOS Y TECNOLOGÍAS IMPLICADAS

Este capítulo contiene:

- Conceptos necesarios para entender el trabajo.
- Tecnologías implicadas, detallando las librerías HitMap y Controllers.

2.1 CONCEPTOS UTILIZADOS

Los siguientes conceptos aparecen a lo largo del trabajo en múltiples ocasiones. Son tratados con profundidad en diversas partes del trabajo y por lo tanto esta sección realiza una introducción simple de los mismos a modo de contexto.

2.1.1 *Computación heterogénea*

Se denomina computación heterogénea [3] a la realizada sobre sistemas compuestos por distintos tipos de dispositivos (CPUs, co-procesadores de tipo GPU de diferentes arquitecturas, etc.). Al contar con distintos dispositivos, cada uno de ellos con unas características diferentes, son una solución mucho más flexible que los sistemas homogéneos, pues permiten explotar las distintas posibilidades y puntos fuertes de sus dispositivos.

Sin embargo, desarrollar aplicaciones en estos sistemas requiere de técnicas específicas y conocimiento experto de los dispositivos presentes en el sistema. Iniciativas como el estándar OpenCL, mantenido por Khronos Group [1] y apoyado por múltiples fabricantes, buscan facilitan el desarrollo de aplicaciones para sistemas heterogéneos ofreciendo un mecanismo único y genérico, como veremos más adelante.

2.1.2 *Co-procesadores tipo GPU y su arquitectura*

Los co-procesadores de tipo GPU (Unidades Gráficas de Procesamiento) son un tipo de hardware en auge en sistemas heterogéneos como se puede ver en la lista TOP500 [7]. Englobados dentro de las unidades de cómputo de tipo "many-cores" (muchos cores) constan de núcleos de menor potencia que los procesadores comunes pero cuentan con muchos de ellos divididos en bloques.

Son capaces de ejecutar una instrucción o cálculo sobre sobre distintos datos de forma paralela (modelo SIMD: Single Instruction, Multiple Data). Tienen su propia memoria DRAM (Dynamic Random-Access Memory) compartida por todos los bloques de cores. Cada bloque de cores tiene una memoria compartida mucho más pequeña y rápida. Cada core dentro de un bloque tiene su propia memoria privada, aún más pequeña y rápida.

2.1.3 *Computación y transferencia de datos*

Las dos tareas principales que se ejecutan en un co-procesador son la computación y la transferencia de datos. Una computación es la ejecución de una o más instrucciones por parte del co-procesador sobre un conjunto de datos. La transferencia de datos implica el movimiento de los mismos desde el *host* o máquina anfitriona al co-procesador o viceversa. Un ejemplo típico de una aplicación paralela sobre un co-procesador se compone de tareas de transferencia de datos al co-procesador, tareas de computación sobre esos datos, y tareas de transferencia de datos al host con los resultados de esa computación.

Algunos co-procesadores tienen uno o más sistemas DMA (Direct Access Memory) que permiten ejecutar tareas de transferencia de datos al mismo tiempo que tareas de computación. Esto genera dependencias entre los datos utilizados en las distintas tareas. Estas dependencias pueden gestionarse en modelos síncronos o asíncronos con una variedad de técnicas, algunas de las cuales son presentadas y aplicadas a lo largo de este trabajo.

2.1.4 *Sincronía y asincronía*

Un modelo de ejecución de tareas síncrono implica el bloqueo de la ejecución del programa hasta que se complete la tarea en cuestión. En el caso antes mencionado una transferencia de datos síncrona solo permite continuar con la ejecución del programa cuando se haya completado. La misma definición vale para las tareas de computación. Esto evita las dependencias entre tareas al no permitir el solapamiento de las mismas.

Un modelo de ejecución de tareas asíncrono permite continuar con la ejecución del programa principal mientras la tarea se ejecuta en segundo plano (en otro hilo, en el co-procesador, etc). En el ejemplo anterior el lanzamiento de una tarea de transferencia de datos en el dispositivo permitiría seguir con la ejecución del programa principal inmediatamente y sin esperas. Los modelos asíncronos incluyen puntos de sincronización. Puntos concretos del código o funciones especiales para verificar que las tareas se han completado y los resultados están disponibles o, si no han terminado dichas tareas, bloquear la ejecución hasta que se completen.

En los algunos modelos de ejecución de tareas asíncronos, aquellos que permiten el solapamiento de tareas, hay que implementar mecanismos de gestión de dependencias entre los datos participantes en las mismas. No es trivial, pero permite mejoras importantes de rendimiento.

2.1.5 *Solapamiento de tareas*

Los modelos de ejecución de tareas asíncronos permiten el solapamiento de las mismas. Actualmente los co-procesadores son capaces de realizar tareas de transferencia de datos al mismo tiempo que tareas de computación. Esto requiere de una programación más costosa y una serie de riesgos sobre la dependencia entre tareas que trataremos a continuación. Algunos co-procesadores que cuenten con la tecnología necesaria son capaces incluso de solapar tareas de transferencia de datos del host al co-procesador con tareas de transferencia de datos del co-procesador al host.

Utilizar un modelo de programación que permita el solapamiento de tareas es uno de los mecanismos de mejora de rendimiento que pueden explotar las aplicaciones paralelas. Si bien la gestión de estos solapamientos, ya sea entre varias tareas de transferencia de datos, o entre tareas de transferencia y tareas de computación, conlleva cierta sobrecarga en el tiempo de ejecución respecto a cuando éstas se realizan de síncrona, la ganancia de rendimiento es más que notable.

Un modelo que utilice solapamiento y una aplicación que permita dicho solapamiento reduce su tiempo de ejecución. Por ejemplo, un caso en que se considere una tarea de computación y una tarea de transferencia de datos de la misma duración las cuales pasan de ejecutarse de forma secuencial a solaparse, reduce su tiempo total de ejecución, como se muestra en la figura 1.

Cuando ocurre solapamiento de tareas aparecen posibles dependencias entre los datos participantes en las tareas. Solo pueden solaparse las tareas que no tengan dependencias entre ellas, por eso en la figura 1 aparece un hueco en el modo asíncrono en las tareas de transferencia entre la tarea 1 y la tarea 5. La tarea 5 tiene una dependencia con la tarea 4 y tiene que esperar a que se complete para iniciar su ejecución.

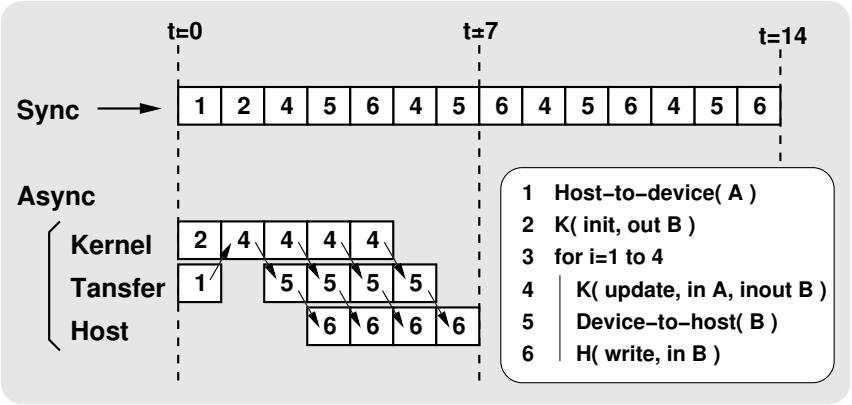

Figura 1: Ejemplos de modos de ejecución síncrono y asíncrono de una secuencia de operaciones en un programa en el co-procesador. El modo asíncrono del ejemplo muestra el solapamiento de operaciones de transferencia de datos (host-to-device o device-to-host), con ejecuciones de kernel en el dispositivo (K), y ejecuciones de funciones en el host (H). Las flechas indican las dependencias entre tareas en los diferentes canales asíncronos. La mejora de rendimiento del modo asíncrono comparado con el modo síncrono es de $S = 2$.

2.1.6 *Dependencia de datos*

Resulta evidente que no se puede ejecutar una computación sobre un conjunto de datos que aún no ha sido transferido al co-procesador. De la misma forma no puede realizarse la transferencia de los resultados, los datos del co-procesador al host, si la computación que los calcula no ha terminado.

Debido a esto, el solapamiento de tareas solo es posible cuando no hay dependencia entre los datos que participan en las mismas. Partiendo del ejemplo básico mostrado en la definición de tareas de computación y tareas de transferencia de datos, se considera ahora un bucle de ejecución en torno a esas 3 tareas (transferencia de datos al co-procesador, computación sobre los mismos, y transferencia de datos resultantes del co-procesador al host).

En cada iteración de ese bucle será posible solapar la transferencia de nuevos datos al co-procesador con el cómputo y la transferencia de los resultados del co-procesador al host. Por lo tanto, también es posible solapar la transferencia de resultados del co-procesador al host con la transferencia del siguiente conjunto de datos del host al co-procesador y la computación sobre esos nuevos datos.

2.2 TECNOLOGÍAS IMPLICADAS

A continuación se introducen las principales tecnologías relacionadas con este trabajo, ya sea por que su descripción resulta necesaria para la comprensión de su evolución a lo largo del tiempo o porque efectivamente hayan sido utilizadas para la implementación actual del mismo.

2.2.1 *CUDA*

CUDA (Compute Unified Device Architecture) es el modelo y la arquitectura en la que la empresa NVIDIA [10] basa sus co-procesadores para cómputo paralelo. Tiene su propio compilador y una serie de herramientas con las que realizar de forma nativa todo el proceso de desarrollo de aplicaciones para sus co-procesadores. Tiene interfaces de programación en distintos lenguajes tales como C/C++, Python, Java, etc.

El modelo de arquitectura de CUDA (Figura 2) define los bloques de *cores* de sus GPUs como *Stream Multiprocessors* (SM's) mientras que cada *core* individual es un *Single Processor* (SP). Así, un SM esta compuesto por un conjunto de SP's con una memoria conjunta, que CUDA denomina memoria compartida (*shared memory*).

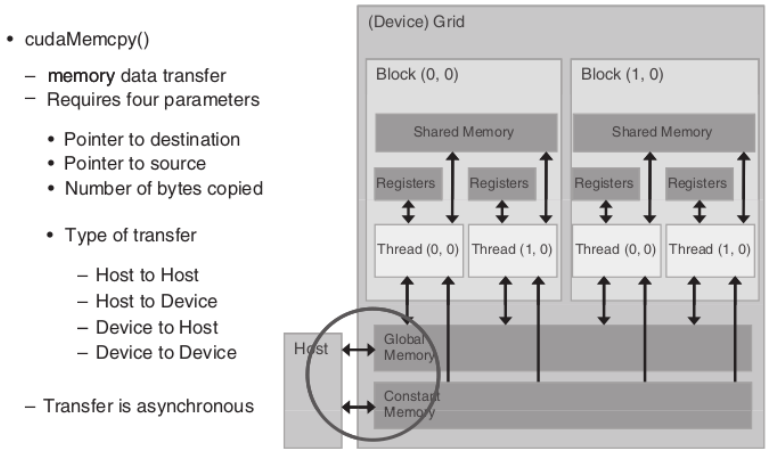

Figura 2: Modelo CUDA para la gestión de memoria.

Los co-procesadores GPU basados en CUDA admiten todas las funcionalidades importantes para este trabajo, tales como la memoria *pinned* (explicada más adelante), los modelos de ejecución asíncronos o el solapamiento de tareas. Además, la versión actual de *Controllers* cuenta con una implementación nativa en CUDA para dar soporte a este tipo de dispositivos.

2.2.2 *OpenCL*

OpenCL [1] es un estándar para desarrollar aplicaciones paralelas en dispositivos heterogéneos. Como estándar simplemente define una interfaz de programación, pero no provee una implementación de la misma para las distintas arquitecturas, sino que éstas son provistas por los fabricantes que deciden añadir soporte para OpenCL a sus dispositivos. OpenCL está soportado por los dos grandes fabricantes de co-procesadores de tipo GPU, NVIDIA [10] y AMD [2].

Por un lado, el estándar define un lenguaje propio, basado en C99 y que se utiliza implementar las tareas de computación, que toman la forma de funciones de código llamadas *kernels*. Por otro lado, define también una interfaz de programación en el *host* que ofrece funciones para construir y compilar esos kernels, gestionar colas de ejecución de tareas (que OpenCL llama comandos), gestionar datos en el co-procesador (que OpenCL llama dispositivo), gestionar movimientos de datos y tareas de cómputo en las colas, o gestionar dependencias entre esas tareas en base a eventos de OpenCL, entre otras. Un detalle de la arquitectura propuesta por OpenCL puede verse en la figura 3.

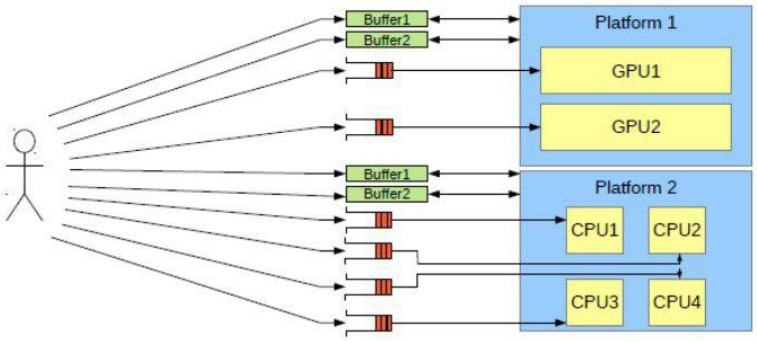

Figura 3: Arquitectura OpenCL para control de uno o varios dispositivos.

OpenCL ofrece dos funcionalidades especialmente relevantes para este trabajo:

- **Gestión de regiones de memoria *pinned***: La memoria *pinned* es un tipo de memoria en el host bloqueada o fija, lo que significa que siempre está cargada en la memoria principal (RAM) y eso hace que su uso para lectura y escritura sea mucho más eficiente. Para conseguir un solapamiento de tareas razonable es necesario el uso de este tipo de memoria en las operaciones de transferencia de datos. Las operaciones de transferencia de datos sobre memoria pinned son mucho más eficientes independientemente del modelo utilizado (sincrono, asíncrono, con o sin solapamiento).

- **Gestión de eventos**: Un evento de OpenCL guarda el estado de una tarea. Las tareas que se añaden a las colas de comandos de OpenCL generan un evento con el que monitorizar su estado cuando utilizamos tareas asíncronas. Las tareas que se añaden a la cola de OpenCL pueden recibir una lista de eventos por los que esperar por su terminación antes de ejecutarse. Los eventos serán la base para gestionar las dependencias entre datos dentro del modelo propuesto en este trabajo. Existe además la posibilidad de crear eventos de usuario, no creados automáticamente por el entorno de ejecución para su propia gestión de las tareas, pero que pueden añadirse a la lista de espera de las mismas. Esto permite una gran flexibilidad a la hora de gestionar las dependencias entre las diferentes tareas, y por lo tanto, asegurarse un correcto tratamiento de posibles riesgos de datos introducidos por éstas en ejecuciones asíncronas.

2.2.3 *Librería HitMap*

Desarrollada por el Grupo Trasgo, la librería HitMap [4] ofrece funcionalidades para el particionado de datos (técnica llamada *tiling*, siendo un *tile* cada una de las partes generadas por dicha técnica) y mapeo de los mismos en la jerarquía de memoria disponible de formas diversas y eficientes. Proporciona una abstracción mediante una interfaz para el lenguaje de programación C que permite generar código independiente del particionado y mapeo escogido. Permite además particionar estructuras de datos dispersas, como grafos o compactas, como arrays y matrices.

Este particionado está diseñado para su uso con el modelo de paralelismo SPMD (Single Program, Multiple Data). Se diferencia del modelo SIMD utilizado en co-procesadores en que en lugar de realizar de forma paralela la ejecución de un cómputo (sea una instrucción o un kernel, por ejemplo) como hace el modelo SIMD, en el modelo SPMD se ejecutan en paralelo los programas enteros, y solo se sincronizan en aquellos puntos especificados por el programador, pudiendo no haber ninguno de estos puntos. El modelo SPMD se utiliza frecuentemente con técnicas de programación distribuida, por lo que las distintas ejecuciones del programa no tienen por qué coincidir en la misma máquina.

En HitMap se pueden encontrar 3 módulos o conjuntos de funcionalidades:

- Funcionalidades de *tiling*, que pueden ser usadas de forma independiente al resto y que *Controllers* utiliza como soporte para sus estructuras de datos. Proporcionan modos de particionado, distribución y acceso eficiente a esos mismos datos.
- Funcionalidades de mapeo, distribución y disposición de datos (layout) en base a una topología virtual.
- Funcionalidades de comunicación para *tiles* distribuidos: Utilizando MPI, permite distribuir en distintos programas, como se indica anteriormente, cada uno de los *tiles* gestionando las comunicaciones entre ellos (sincronización) de forma transparente al usuario. Soporta la creación de patrones de comunicación dependientes del mapeo del *tile*.

2.2.4 *Librería Controllers*

Controllers [8] es una librería creada por el Grupo Trasgo para el desarrollo de aplicaciones paralelas de forma unificada sobre sistemas heterogéneos. Ofrece un conjunto de funcionalidades que son comunes a los modelos de múltiples plataformas de paralelismo sobre co-procesadores junto con una implementación nativa de esas funcionalidades en función de las capacidades sistema sobre el que se compile. También ofrece una gestión automática de las dependencias entre los datos participantes en cada tarea completamente transparente al usuario.

Proporciona la entidad abstracta *controlador*, que permite al programador manejar fácilmente el lanzamiento de tareas de transferencia de datos y computación a un co-procesador o dispositivo. Consta con una cola de tareas propia y varios modelos de gestión de las mismas. La arquitectura de *Controllers* puede verse en la Figura 4.

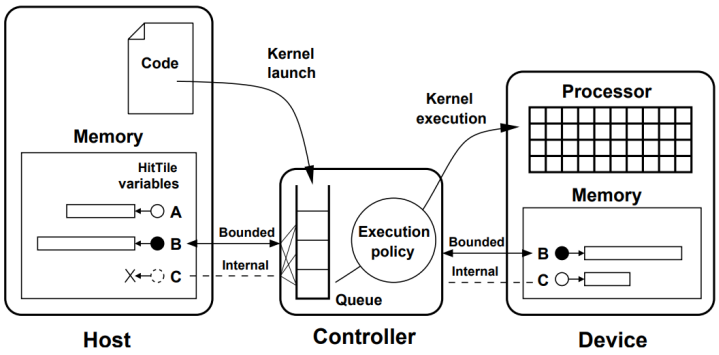

Figura 4: Arquitectura de Controllers.

Algunas de sus principales funcionalidades son:

- Permite definir "kernels" (tareas de computación) de forma unificada, utilizando el mismo código para distintas arquitecturas. También permite definir kernels específicos de una arquitectura para que los usuarios expertos en una tecnología concreta como CUDA o OpenCL puedan sacar el máximo rendimiento a las tareas de computación. Estos kernels específicos utilizan el mismo lenguaje unificado que los genéricos pero admiten código único de las tecnologías nativas para las que van destinados.
- Soporta los modelos síncrono y, recientemente, un prototipo de modelo asíncrono de ejecución que permite solapamiento [11]. Dicho modelo asíncrono permite solapamiento entre las tareas, gestionando de forma transparente las dependencias entre datos de las mismas. Sin embargo, actualmente este modelo solo está soportado sobre CUDA. Este trabajo propone una mejora del mismo junto con una implementación adicional en OpenCL que permita su uso sobre sistemas heterogéneos.

- Gestiona la memoria en los co-procesadores, incluidas tareas de copia de datos (transferencia de datos del host al co-procesador o viceversa). Incluye tareas de sincronización para realizar esperas en el modo asíncrono. En el modo síncrono no son necesarias.

2.2.5 *CMake*

CMake es una familia de herramientas para construir, probar y compilar software. Permite un control de la compilación del software independiente de la plataforma y compilador subyacente. Genera Makefiles nativos (en el caso de sistemas Unix). Su único requisito es un compilador de C++. Es especialmente útil para gestionar configuraciones y compilaciones complejas, en proyectos grandes o con módulos opcionales. *Controllers* utiliza CMake para su configuración y compilación.

3

DISCUSIÓN DEL MODELO

Este capítulo contiene:

- Gestión de tareas en *Controllers*.
- Nuevo modelo asíncrono basado en eventos.
- Discusión del nuevo modelo propuesto.

3.1 GESTIÓN DE TAREAS EN CONTROLLERS

En este apartado se introducen los diferentes modelos de gestión de tareas en *Controllers*, definiendo además los tipos de tareas que proporciona la librería y el mecanismo que permite el solapamiento de las mismas. En la figura 5 se puede ver la arquitectura de *Controladores* y el modelo de gestión de tareas asíncrono con solapamiento.

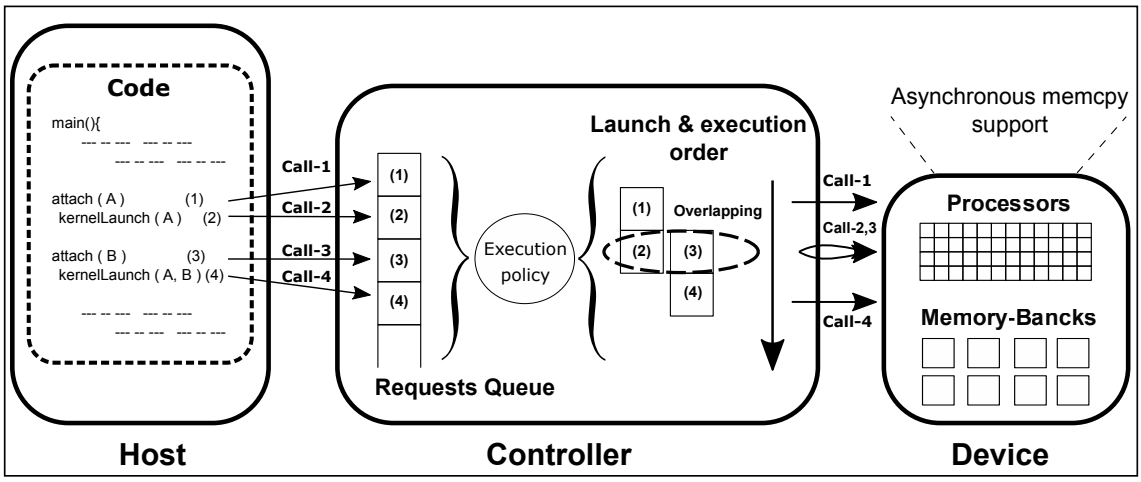

Figura 5: Controllers: Solapamiento de tareas mediante políticas de ejecución.

3.1.1 *Políticas de gestión de tareas*

Controllers permite el solapamiento de tareas de computación y transferencia de datos de forma transparente para el usuario. Que sea transparente implica que el usuario no debe modificar el código de sus programas para conseguir este solapamiento. Estas tareas pueden gestionarse de forma síncrona (sin solapamiento) o asíncrona (con solapamiento), de acuerdo con lo que se define como *política* del controlador. Esta política puede fijarse internamente en el código o a través de una variable de entorno del sistema.

3.1.2 *Tipos de tareas en Controladores*

Controllers define una serie de funciones en su interfaz para su uso, añadiendo cada una de estas funciones una tarea específica en la cola interna del controlador. Estas tareas pueden dividirse en 3 categorías: de computación, de transferencia de datos y de sincronización. Así mismo, define también una serie de funciones auxiliares para gestión de *tiles* las cuales no se detallan por no ser objeto de este trabajo.

- **Tareas de computación:** Para las tareas de computación Controladores define launchKernel. Recibe un nombre de un kernel definido previamente por el usuario, y los tiles ordenados que intervienen en el mismo, en el orden que el usuario haya definido previamente. La definición de kernels ha sido tratada ampliamente en trabajos anteriores citados y no entra dentro del contexto de este trabajo. las funciones de computación son asíncronas(no bloquean la ejecución) en ambas políticas.

- **Tareas de transferencia de datos:** Para las tareas de transferencia de datos Controladores define dos funciones, moveTo y moveFrom. moveTo para transferencia de datos del host al co-procesador y moveFrom para transferencia de datos del co-procesador al host. Estas funciones reciben el tile sobre el que realizar la transferencia. Estas transferencias son síncronas o asíncronas en función de la política escogida. En el caso de la política asíncrona, es necesario el uso de funciones de sincronización.

- **Tareas de sincronización:** Para las tareas de sincronización Controladores define otras dos funciones, waitTile y globalSync. Estas funciones son necesarias únicamente en la política asíncrona. Pueden ser usadas con una política síncrona, aunque no tendrán un efecto real ya que la ejecución en política síncrona siempre está sincronizada y no hay tareas por las que esperar por su terminación. waitTile permite esperar por las tareas de transferencia de cualquier tipo asociadas a un tile, que recibe como parámetro. Mediante globalSync se define un punto de sincronización global, que bloquea la ejecución principal hasta que se completan

todas las tareas añadidas a la cola hasta ese momento.

3.1.3 *Hilos de ejecución de Controladores*

Controllers utiliza dos hilos de ejecución que, en líneas generales, se encargan respectivamente de ejecutar el código de usuario y de gestionar la cola de tareas:

1. El hilo de *host* es el hilo principal del programa y se encarga de ejecutar el código creado por el usuario. El usuario en ese hilo, utilizando las funciones que define Controladores, añade tareas a la cola interna del controlador. Estas tareas, como se ha indicado anteriormente, pueden ser de movimiento de datos, lanzamiento de kernels o sincronizaciones.
2. El otro hilo de ejecución, denominado hilo del controlador, se encarga de desencolar tareas, evaluarlas y ejecutarlas.
   - En la política síncrona se evalúan y ejecutan las tareas una a una por orden y de forma secuencial, por lo que no es necesario gestionar las dependencias entre los datos.
   - En la política asíncrona actual, al evaluar una tarea el controlador debe decidir si ésta pueden ejecutarse inmediatamente o si hay alguna dependencia incumplida en los datos que manipula. Si es así, espera a que estas dependencias se cumplan antes de lanzar la tarea, bloqueando el hilo de ejecución. Ese hilo es desbloqueado por una función de retrollamada (*callback*) que es utilizada por otro hilo auxiliar (propio del co-procesador, la política asíncrona previa a este trabajo está implementada solamente para código CUDA sobre GPUs NVIDIA) cuando la tarea de la que depende la actual tarea en evaluación finaliza.

3.1.4 *Tipos de datos en el co-procesador según su uso y las dependencias que generan*

Los datos que utiliza una tarea de computación (kernel) pueden ser de tres tipos, de entrada (IN), de salida (OUT) o de entrada/salida (IO). Una tarea de computación genera dependencias para los tiles que participan en ella en base a sus tipos.

Los datos que utilizan las tareas de transferencia no tienen tipo en sí, pero generan dependencias en base a la dirección de la transferencia (host al co-procesador o viceversa).

3.1.5 *Dependencias generadas por tiles en las tareas de transferencia*

Las dependencias entre los datos participantes en las tareas de tipo transferencia de datos se muestran en la figura 6 y se decriben a continuación:

- Tareas de transferencia del host al co-procesador (tareas moveTo): Para sobrescribir los datos en el co-procesador, es necesario esperar por aquellas tareas de computación que utilicen dichos datos como entrada (IN) o como entrada y salida (IO) para evitar que el kernel acceda a esos nuevos datos en lugar de a los que aún está manipulando. También tienen que esperar por las tareas de transferencia desde el co-procesador, para no sobrescribir datos que ya hayan sido computados pero aún no hayan sido transferidos de vuelta al host.
- Tareas de transferencia del co-procesador al host (tareas moveFrom): Para poder recuperar los datos desde el co-procesador es necesario esperar por aquellas tareas de computación que los utilicen como salida (OUT) o como entrada y salida (IO), para evitar copiar versiones intermedias cuyo cómputo aún no haya finalizado. También tienen que esperar por las tareas de transferencia al co-procesador, si bien este es un caso hipotético (recuperar datos que no han sido utilizados) que se ha preferido contemplar para asegurar la integridad de los datos en el modelo.

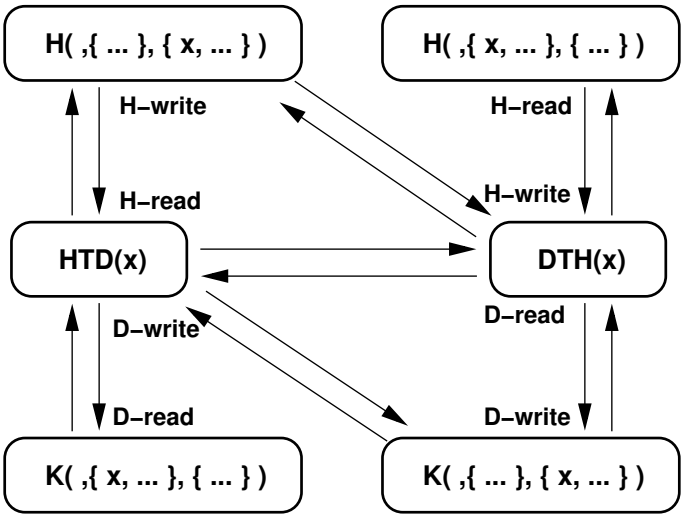

Figura 6: Dependencias entre las operaciones debido a los roles de escritura y lectura de los parámetros en el host o en el dispositivo. Para simplificar, la operación Wait no se muestra en la figura. Depende de de todas las demás operaciones con el mismo parámetro.

3.1.6 *Dependencias generadas por tiles en las tareas de computación*

Las dependencias entre los datos que generan las tareas de computación (kernelLaunch) para cada una de las estructuras de datos (*tiles*) implicadas en la ejecución de un *kernel* dado dependen de su tipo (IN, OUT, IO) y provocan esperas por:

- Tile de tipo IN: Para que un kernel pueda utilizar los datos como entrada, es necesario esperar por todas aquellas tareas anteriores de la cola que los modifiquen. Un tile de tipo IN provoca una espera por las tareas de transferencia del host al co-procesador anteriores, y por las tareas de computación que utilicen el tile como OUT o como IO.
- Tile de tipo OUT: Para que un kernel pueda modificar los datos, es necesario esperar por todas aquellas tareas anteriores de la cola que los utilicen como entrada. Un tile de tipo OUT provoca una espera por las tareas de transferencia del co-procesador al host anteriores, y por las tareas de computación que utilicen el tile como IN o como IO.
- Tile de tipo IO: Los tiles de tipo IO provocan esperas como si fueran de los dos tipos anteriores (IN y OUT). Por lo tanto la tarea de computación espera por las tareas anteriores de transferencia de host al co-procesador y viceversa, y por las tareas anteriores de computación que utilicen el tile, para todos los tipos posibles (IN, OUT e IO).

3.1.7 *Solapamiento de tareas*

En la aproximación inicial de *Controllers* a la gestión asíncrona de tareas, el soporte para el solapamiento de tareas se limitaba a GPUs NVIDIA programadas en CUDA, estando por ello implementado mediante llamadas asíncronas a la librería nativa del co-procesador.

El hilo del controlador extrae tareas de la cola y las evalúa en el mismo orden en el que se encolaron. Si una tarea no depende de ninguna anterior se ejecuta inmediatamente y se extrae la siguiente de la cola. En cambio, en caso de que exista alguna dependencia en la tarea evaluada, el hilo del controlador se bloquea hasta que la tarea de la que depende le despierte. Al estar bloqueado no se siguen evaluando nuevas tareas, y no puede producirse solapamiento con las tareas en la cola hasta que no se cumplan esas dependencias de la tarea en evaluación.

Esta aproximación inicial al modelo asíncrono consigue efectivamente solapamiento de tareas, ya que las tareas evaluadas inician su ejecución en el momento en el que sus dependencias se cumplen. Sin embargo, no evalúa nuevas tareas hasta que no se cumplen las dependencias de la tarea en evaluación con las anteriores, lo que supone una carencia importante que precisamente este trabajo viene a resolver.

3.2 MODELO ASÍNCRONO BASADO EN EVENTOS

A continuación se describe el nuevo modelo asíncrono, detallando las mejoras que introduce frente al anterior y presentando el enfoque basado en eventos que utiliza este nuevo modelo.

3.2.1 *Problemas del modelo anterior*

El modelo asíncrono inicial, a pesar de conseguir el solapamiento de tareas junto con el incremento de rendimiento que ello implica, presenta una carencia importante, y es que debido al bloqueo del hilo del controlador, la evaluación de tareas se detiene. Esto imposibilita el potencial solapamiento de tareas posteriores.

3.2.2 *Enfoque del nuevo modelo como un problema de múltiples lectores y escritores*

El nuevo modelo propuesto pretende solucionar esta pérdida potencial de solapamiento mediante una nueva aproximación a la gestión de dependencias. Poniendo el foco en las estructuras de datos y no en las tareas, la gestión de dependencias se convierte en una versión del clásico problema de concurrencia entre múltiples lectores y múltiples escritores.

El problema de múltiples lectores y escritores es bien conocido en el ámbito de los sistemas concurrentes. Múltiples actores (hilos) quieren utilizar la misma estructura de datos, ya sea para leerla o para escribir en ella (lectores y escritores). No pueden acceder todos a la vez porque podrían darse inconsistencias en esos datos: mientras un hilo actualiza los datos, otro los está leyendo y no se puede garantizar que los datos leídos sean los iniciales, los actualizados, o una mezcla de ambos.

3.2.3 *Solución al problema de múltiples lectores y escritores en el nuevo modelo*

Para solucionar el problema de múltiples lectores y escritores es necesario disponer de mecanismos de sincronización que permitan establecer un orden en la ejecución de tareas concurrentes.

Al ver la gestión de dependencias del modelo de ejecución asíncrono con solapamiento de tareas como un problema de lectores y escritores sobre las estructuras de datos (tiles) resulta posible calcular en tiempo de ejecución las dependencias de cara a las tareas posteriores.

De esa manera se puede continuar con la evaluación de tareas y la gestión de las dependencias de las mismas incluso sin que las tareas anteriores de las que dependa la tarea en cuestión hayan iniciado su ejecución.

3.2.4 *Ventajas del nuevo modelo*

La principal ventaja derivada de eliminar la necesidad de bloquear el hilo de Controladores que evalúa las tareas y permitir que se sigan evaluando tareas de la cola, y teniendo en cuenta que una tarea evaluada se ejecuta en el momento en el que sus dependencias se cumplen, este modelo permite el máximo solapamiento posible entre todas las tareas introducidas en la cola.

3.2.5 *Modelo de gestión de las dependencias entre tiles basado en eventos*

Esta nueva gestión de dependencias se basa en eventos. En este modelo, un evento es una entidad abstracta que guarda información del estado de un *tile* en alguna de las operaciones soportadas por éstos (lectura, escritura).

Los eventos, por tanto, indican en todo momento si el *tile* está disponible para lectura o para escritura de forma independiente, en base a las dependencias que generen las tareas. Las tareas, al evaluarse, fijan como dependencia el evento y esperan para iniciar su ejecución a que los eventos de los que dependan se cumplan. De esta manera se asegura la consistencia entre datos a la vez que se consigue establecer las dependencias entre tareas de forma no bloqueante, pudiendo continuar con la evaluación de tareas.

3.2.6 *Primera aproximación: dos eventos por tile*

La primera aproximación para utilizar eventos en la gestión de dependencias utiliza dos eventos por tile, uno para lectura y otro para escritura. Las tareas que requieren del tile para leerlo tienen que esperar al último evento de escritura presente. Las tareas que requieren del tile para escribirlo tienen que esperar al último evento de lectura del tile.

No es casual que siempre se espere por el último evento, tanto de lectura como de escritura. Esto permite al modelo actualizar los últimos eventos de cada tipo, sin necesidad de guardar varios eventos a modo de histórico de los mismos.

Una tarea, en el momento de su evaluación, realiza una copia temporal de los eventos por los que tiene que esperar (sus dependencias) y actualiza el evento del tile correspondiente, generando las futuras dependencias para las siguientes tareas.

A partir del momento de evaluación de una tarea ninguna tarea posterior necesita en ningún caso los eventos anteriores que esa tarea haya generado, puesto que las tareas tienen un orden de ejecución. Por lo tanto esos eventos copiados por la tarea en evaluación y que ya no están disponibles, pueden ser eliminados de forma segura al finalizar la ejecución de la tarea en evaluación.

El solapamiento se produce entre tareas de transferencia de datos y tareas de computación. Actualmente muchos co-procesadores soportan transferencias de datos en ambos sentidos sobre el mismo bus PCI-e, lo que permite solapar tareas de transferencia de datos en sentidos inversos entre si. Para garantizar la consistencia de la ejecución del programa, el orden de ejecución de tareas entre las que haya dependencias debe ser el mismo orden en el que llegan a la cola interna del controlador.

Para asegurar esto, todas las tareas del mismo tipo (transferencia de datos del host al co-procesador, transferencia de datos del co-procesador al host, y computación) deben ejecutarse de forma secuencial y ordenada. Las dependencias se producen entre tareas de tipos diferentes y son tratadas en el momento de la evaluación de una tarea.

3.2.7 *Problema de la aproximación con dos eventos por tile*

Hay determinados casos particulares donde no se cumplen las aserciones expuestas en la sección anterior. A continuación se plantea uno de estos contraejemplos.

Sean las siguientes tareas, todas ellas sobre el mismo tile:

1. Tarea de computación con el tile como entrada (IN) que denominaremos K.
2. Tarea de transferencia de datos del co-procesador al host que denominaremos DtH.
3. Tarea de transferencia de datos del host al co-procesador que denominaremos HtD

Estas tres tareas se envían a la cola interna del controlador por el usuario siguiendo exactamente este orden. De acuerdo con la aproximación con dos eventos, la evaluación de las tareas y la actualización de sus eventos y dependencias se realizará como se describe a continuación:

1. La tarea de computación con el tile como entrada (IN) denominada K se evalúa. Al no haber tareas previas no encuentra ningún evento por el que esperar. Genera un evento de lectura sobre el tile y comienza su ejecución.
2. La tarea de transferencia de datos (tile) del co-procesador al host denominada DtH se evalúa. Comprueba los eventos previos de escritura sobre el tile, pero como éstos no existen no encuentra ninguna dependencia por la que esperar. Genera un evento de lectura sobre el tile sobrescribiendo el de la tarea K y se ejecuta.

3. La tarea de transferencia de datos (tile) del host al co-procesador denominada HtD se evalúa. Comprueba los eventos de lectura previos sobre el tile, encontrando el generado por la tarea DtH, por lo que lo copia como dependencia. Genera un evento de escritura sobre el tile y comienza su ejecución. Al ser un modelo de ejecución asíncrono no iniciará su ejecución real hasta que se cumplan sus dependencias, pero la evaluación de la tarea termina.

En este punto es necesario fijarse en el momento de finalización de las tareas. Si la tarea DtH termina su ejecución después de que termine la ejecución de la tarea K, entonces el modelo funciona correctamente.

Sin embargo, si la tarea DtH termina antes de que finalice la ejecución de la tarea K, la tarea HtD iniciará su ejecución real y empezará a escribir en el tile antes de que la tarea K termine de leer el mismo tile, y el modelo no puede garantizar la consistencia de los datos.

El mismo problema aparecería en el caso complementario, basado en dependencias de escritura. Por tanto, no se puede considerar válido el modelo de ejecución asíncrona basado en dos eventos por tile, puesto que no garantiza en todos los casos la consistencia de datos y operaciones.

3.2.8 *Aproximación final: cuatro eventos por tile*

La solución al problema presentado en la sección anterior es utilizar dos eventos más en cada tile. Dos de esos eventos (uno de lectura y otro de escritura) son generados y actualizados por las tareas de transferencia de datos. Los otros dos eventos (uno de lectura y otro de escritura) son generados y actualizados por las tareas de computación.

Las tareas que encuentren dependencias en un tile necesitarán esperar por todos los eventos de cada tipo (lectura y/o escritura). Cada tarea actualiza sus eventos relacionados (lectura y/o escritura) pero solo aquellos pertenecientes a su tipo de tarea (computación o transferencia).

Siguiendo el ejemplo de la sección anterior, con las 3 tareas descritas, la evaluación y actualización de eventos de esa secuencia se realizaría de la siguiente forma:

1. La tarea de computación con el tile como entrada (IN) denominada K se evalúa. Al no haber tareas previas no encuentra ningún evento por el que esperar. Genera un evento de lectura de computación sobre el tile y comienza su ejecución.
2. La tarea de transferencia de datos (tile) del co-procesador al host denominada DtH se evalúa. Comprueba los eventos de escritura sobre el tile previos, que no existen, por no que no encuentra ninguna dependencia por la que esperar. Genera un evento de lectura de transferencia sobre el tile y comienza su ejecución.
3. La tarea de transferencia de datos (tile) del host al co-procesador denominada HtD se evalúa. Comprueba los eventos de lectura sobre el tile previos, y encuentra el que ha generado la

tarea K y el que ha generado la tarea DtH por lo que los copia como dependencias. Genera un evento de escritura de transferencia sobre el tile y comienza su ejecución.

De esta forma se elimina el problema de sobrescribir eventos del mismo tipo de uso (lectura y escritura) pero diferentes tipos de tarea (transferencia y computación). En este ejemplo la tarea HtD tiene como dependencia ambas tareas anteriores, por lo que no iniciará su ejecución real hasta que ambas finalicen, independientemente de cuál de esas tareas (K o HtD) sea la más larga.

Se añade también una dependencia en aquellas tareas que modifiquen los tiles en el co-procesador (transferencias de host al co-procesador y computación con el tile como OUT o IO), para que las modificaciones sobre un tile no se realicen al mismo tiempo.

En un caso real raramente puede tener lugar la situación anterior, y de aparecer generalmente es debida a un error del usuario respecto del orden en el que ha introducido las tareas en la cola. Este caso supone que o bien los datos resultantes de una computación se descartan, o bien los nuevos datos escritos en el tile desde el host se descartan antes de ser usados nuevamente. Si bien no se ha encontrado ninguna situación real donde esto sea necesario, esto no implica necesariamente que ésta no exista. Por ello, finalmente se ha optado por añadir la dependencia anteriormente descrita para garantizar la consistencia de datos.

Así, este modelo de ejecución asíncrona basado en cuatro eventos por tile efectivamente garantiza la consistencia de datos y el orden de operaciones, y por lo tanto se considera válido para su incorporación a *Controllers*. En el capítulo 5 se detallan las pruebas de validación realizadas sobre esta versión del modelo.

3.2.9 *Nomenclatura y definición de los eventos*

La nomenclatura de los diferentes conceptos del modelo se ha construido desde la perspectiva del dispositivo que realiza efectivamente la computación. Así, una operación de transferencia de datos del host al dispositivo será una **escritura** en el tile en el dispositivo; y viceversa, una transferencia de datos del co-procesador al host será una **lectura** del tile en el dispositivo.

Por un lado, para las tareas de transferencias de datos de host al dispositivo o viceversa, que se realicen sobre el tile, se podrán registrar los siguientes eventos:

1. Para las tareas de transferencia del host al dispositivo, un evento llamado offloading_write, para la escritura por el host en el tile en el dispositivo. Offloading en este contexto significa transferencia de datos a otro dispositivo o plataforma.
2. Para las tareas de transferencia del dispositivo al host, un evento llamado offloading_read, para la lectura por el host del tile en el dispositivo.

Por otro lado, para las tareas de computación y en base al tipo que tenga el tile en el kernel, se podrán registrar otros dos eventos:

1. Para las tareas de computación que utilicen el tile como salida (OUT) o entrada / salida (IO), un evento llamado kernel_write, para la escritura por el kernel del tile en el dispositivo.
2. Para las tareas de computación que utilicen el tile como entrada (IN) o entrada / salida (IO), un evento llamado kernel_read, para la lectura por el kernel del tile en el dispositivo.

3.2.10 *Estructura final de las dependencias entre tiles*

Respecto a las tareas de transferencia de datos, los eventos por los que éstas han de esperar y los eventos que éstas actualizan son los siguientes:

- Tareas de transferencia del host al dispositivo (tareas moveTo): Para escribir en el tile, es necesario esperar por los eventos de lectura (kernel_read y offloading_read) del tile. Además para garantizar la consistencia de datos, es necesario esperar por los eventos de escritura del tile (kernel_write y offloading_write). Las tareas de transferencia del host dispositivo actualizan el evento de offloading_write.
- Tareas de transferencia del dispositivo al host (tareas moveFrom): Para leer del tile, es necesario esperar por los eventos de escritura (kernel_write y offloading_write) del tile. Las tareas de transferencia del dispositivo al host actualizan el evento de offloading_read.

En cuanto a las tareas de computación, los eventos por los que han de esperar de cada uno de los tiles en base a su tipo (IN, OUT o IO) y los eventos que actualizan son los siguientes:

- Tile de tipo IN: Para que un kernel pueda leer del tile, es necesario esperar por los eventos de escritura (kernel_write y offloading_write) sobre ese tile. Un tile de tipo IN actualiza el evento kernel_read.
- Tile de tipo OUT: Para que un kernel pueda escribir en el tile, es necesario esperar por los eventos de lectura (kernel_read y offloading_read) sobre ese tile. Además para garantizar la consistencia de datos, es necesario esperar por los eventos de escritura (kernel_write y offloading_write) sobre el tile. Un tile de tipo OUT actualiza el evento kernel_write.
- Tile de tipo IO: Para que un kernel pueda leer y escribir en el tile, es necesario esperar por los eventos de lectura (kernel_read y offloading_read) y escritura (kernel_write y offloading_write) sobre ese tile. un tile de tipo IO actualiza los eventos kernel_read y kernel_write.

3.3 DISCUSIÓN DEL MODELO

El nuevo modelo asíncrono propuesto consigue, al igual que el anterior, solapamiento y gestión automática de las dependencias entre datos de forma transparente. Su principal ventaja respecto a la aproximación previa es que este nuevo modelo basado en eventos evita el bloqueo del hilo de ejecución del controlador. Así, la evaluación de tareas no se detiene innecesariamente continua y

se incrementa su potencial solapamiento al poder considerar todas las tareas de la cola y no solo aquellas que no tengan ninguna dependencia incumplida.

Por otra parte, al pasar de una implementación nativa en CUDA para GPUs NVIDIA a otra para sistemas heterogéneos basada en OpenCL, el soporte del modelo se vuelve mucho más simple y eficiente. La implementación de funciones de *callback* en CUDA es mucho más compleja que el uso de eventos de OpenCL, y la utilización de semáforos para bloquear y desbloquear hilos implica una sobrecarga de tiempo que la copia de eventos evita por completo.

Además, al ser asíncrono y no bloquear el hilo del host, la ejecución de las tareas se solapa con la ejecución de instrucciones (de cálculo, de memoria, etc.) en el host, siempre que los puntos de sincronización estén correctamente dispuestos a lo largo de la ejecución del código de usuario. Esto es necesario en todas las plataformas y tecnologías de paralelismo con ejecuciones asíncronas, por lo que no supone para el usuario necesidad alguna de conocimiento experto adicional sobre *Controllers*, sino que son conceptos que forman parte de los mínimos habitualmente requeridos para el desarrollo de aplicaciones paralelas asíncronas.

Controllers utiliza análisis en tiempo de ejecución de las dependencias entre las tareas. Otros análisis, ya sean en tiempo de ejecución o previos, pueden ser capaces de realizar un reordenamiento de las tareas de la cola, permitiendo un mayor solapamiento de tareas. Esas técnicas pueden ser implementadas sobre este modelo en el futuro, pero quedan fuera del alcance de este trabajo.

Es un modelo válido para los sistemas heterogéneos en auge, incluidos los grandes sistemas de supercomputación, puesto que el único requisito que tiene es que las plataformas que conforman esos sistemas tengan soporte para eventos, como ocurre en el caso de CUDA y OpenCL, lo que cubre la basta mayoría de dispositivos tipo GPU que conforman los grandes sistemas heterogéneos actuales.

El modelo, como conclusión, aporta un modo de ejecución asíncrono que permite solapamiento de tareas sin intervención del programador, lo que proporciona mejoras de rendimiento de forma transparente y sin que el usuario necesite cambiar una sola línea de código de su aplicación.

4

IMPLEMENTACIÓN

Este capítulo contiene:

- Introducción a OpenCL
- Reestructuración de la librería *Controllers*
- Implementación del backend de OpenCL

4.1 INTRODUCCIÓN A OPENCL

Si bien ya fue brevemente introducido en la relación de tecnologías del capítulo 2, existen toda una serie de conceptos propios de OpenCL que resulta imprescindible explicar con cierto detalle para poder comprender cómo se ha implementado en Controllers el soporte para este estándar de computación heterogénea.

OpenCL solamente provee una interfaz, siendo cada implementación de OpenCL responsabilidad de los fabricantes de dispositivos, siendo habitual que existan diferencias de criterio entre ellos a la hora de resolver aquellos detalles que no están claramente especificados en la interfaz. Estos detalles han sido tratados de un modo generalista, añadiendo todo lo necesario para que funcione en los dispositivos que Controladores soporta (co-procesadores de NVIDIA y AMD). La arquitectura propuesta por OpenCL se puede ver de modo general en la figura 3

4.1.1 *Estructuras y funciones de OpenCL*

OpenCL es una tecnología realmente extensa, y dado que excede los límites del presente trabajo, en esta sección no se describirá ni el lenguaje propio de los *kernels* de computación ni los mecanismos asociados de creación, configuración y gestión de los mismos. Sí se tratarán aquellos relacionados con su ejecución, ya que al final del proceso las tareas de computación de Controllers

estarán irremediablemente ligadas a la ejecución de kernels OpenCL.

A continuación se detallan las estructuras y funciones de OpenCL necesarias para Controladores, describiendo de forma general qué son y para qué sirven. En primer lugar, se describen las estructuras más relevantes:

- *Platform*: Una plataforma de OpenCL es una cada una de las implementaciones del estándar disponibles en el sistema. Por ejemplo, un sistema con co-procesadores de NVIDIA y AMD puede tener la implementación de cada uno de los fabricantes. Una plataforma tiene asociados los *dispositivos* que soporta, pudiendo aparecer un dispositivo en más de una plataforma.

- *Device*: Un dispositivo de OpenCL es una abstracción de un elemento hardware (CPUs, GPUs u otro tipo de co-procesadores) que soporta la implementación (plataforma) de OpenCL a la que está ligado.

- *Context*: Un contexto de OpenCL es una estructura que OpenCL utiliza para manejar en tiempo de ejecución otras estructuras comunes a un dispositivo, como las colas de comandos, la memoria o los kernels, que serán descritos más adelante. Los contextos están ligados a uno o varios dispositivos.

- *Program*: Un programa de OpenCL es una estructura con el código fuente que se quiere ejecutar en algún dispositivo. Los programas se construyen (compilan) en tiempo de ejecución. Un programa está ligado a un contexto (y por lo tanto a unos dispositivos).

- *Kernel*: Un kernel de OpenCL es un objeto que contiene la versión compilada (construida) de una función de un programa. Esta ligada a un programa (y por lo tanto a un contexto y unos dispositivos). Posteriormente se describen las funciones necesarias para fijar los argumentos que recibe el kernel y ejecutarlo en un dispositivo concreto.

- *Buffers*: Un buffer de OpenCL es una estructura que referencia a una región de memoria en el dispositivo. Contiene información extra sobre esa región de memoria, como su tipo de uso en el dispositivo (lectura o escritura), quien la utiliza (host, dispositivo o ambos), si es *memoria pinned* o no, etc. Se utiliza para los comandos de transferencia de datos y como argumentos para los kernels.

- *Command queue*: Una cola de comandos de OpenCL es una estructura que maneja la ejecución ordenada de comandos (tareas). Un comando de OpenCL es una tarea (transferencia de datos, computación, sincronización) con unas dependencias (eventos de espera), y un modo de ejecución (síncrono o asíncrono). Un comando siempre genera un evento con el que monitorizar su estado. Para conseguir solapamiento es necesario varias colas de comandos, no se pueden solapar comandos de la misma cola, porque aunque el estándar admita esa posibilidad y se pueda crear una cola de comando con un orden de ejecución no secuencial, las

implementaciones no lo soportan. La única forma de sincronizar comandos entre distintas colas es utilizar los eventos que estos comandos generan. Una cola de comandos esta ligada a un contexto y a un dispositivo concreto (los contextos pueden contener varios dispositivos).

- *Event*: Los eventos de OpenCL son estructuras generadas por los comandos en las que se almacena información sobre el estado de los mismos. Permiten monitorizar la ejecución de los comandos, establecer dependencias entre ellos u obtener datos relativos a la ejecución como los tiempos o los errores, etc. Los eventos están ligados a un contexto, de modo que pueden utilizarse para sincronizar colas de comandos diferentes siempre que éstas pertenezcan al mismo contexto, aunque estén ligadas a distintos dispositivos. Se pueden crear eventos de usuario, ligados a un contexto, y usarlos como dependencias de un comando en una cola con el mismo contexto que el evento. Existen funciones para cambiar el estado de un evento de usuario, por lo que se puede controlar el momento en el que se cumplen las dependencias de un comando.

A continuación, se clasifican y describen las principales funciones de OpenCL utilizadas en esta nueva implementación de *Controllers*:

- *Funciones de gestión de memoria*: Ofrecen mecanismos de creación y destrucción de los buffers, fijando su tipo de uso en el dispositivo (lectura o escritura), el actor u actores responsables de las mismas (host, dispositivo o ambos) o el tipo de memoria a la que apuntan (por ejemplo, si se trata de memoria *pinned*). Algunas funciones específicas para la gestión de memoria de OpenCL son *clCreateBuffer* y *clReleaseMemObject* para la creación y liberación de los mismos.

- *Funciones de gestión de colas*: OpenCL proporciona funciones para la gestión de las colas de comandos. Dos de ellas especialmente importantes son *clFlush* y *clFinish*. La primera asegura que todos los comandos introducidos en una cola de OpenCL serán ejecutados (enviados al dispositivo) en algún momento. La segunda es una función de sincronización a nivel de cola que espera a que se completen todos los comandos pendientes de la misma. Algunas implementaciones específicas de OpenCL requieren de *clFlush*, como la implementación de NVIDIA de OpenCL para sistemas Windows, mientras que para otras no es necesario, como la implementación de NVIDIA de OpenCL para sistemas basados en Linux.

- *Funciones de encolado*: OpenCL proporciona también las funciones necesarias para poder añadir comandos a la cola. Dichos comandos pueden clasificarse en dos grandes grupos:

    1. *Comandos de transferencia de datos*: Para realizar la transferencia de datos en buffers, OpenCL provee de *clEnqueueReadBuffer* y *clEnqueueWriteBuffer* para la transferencia de datos del dispositivo al host, y viceversa. Ambas pueden usarse en modo

síncrono o asíncrono, admitiendo por tanto una lista de eventos por los que esperar y generando un evento que almacena el estado del comando.

2. *Comandos de lanzamiento de kernels*: Para poder ejecutar código en un dispositivo OpenCL es preciso encolar un comando de lanzamiento de kernel, siendo *clEnqueueNDRangeKernel* la función encargada de ello. Los comandos de lanzamiento de kernels siempre son asíncronos, de modo que requieren de mecanismos de sincronización explícita para conseguir que el comportamiento de un programa completo sea el deseado. Estos comandos también admiten una lista de eventos por los que esperar, y generan su correspondiente evento para almacenar el estado de su ejecución.

- *Funciones de sincronización de eventos*: Además de las funciones de sincronización de colas ya comentadas (*clFinish*), OpenCL provee de una función para esperar por una lista de eventos llamada *clWaitForEvents*. Esta función recibe una lista de eventos por los que esperar. Su comportamiento es síncrono, de manera que bloquea la ejecución hasta que todos los comandos asociados a los eventos de esa lista hayan finalizado.

4.2 REESTRUCTURACIÓN DE CONTROLLERS

Una vez discutido en el capítulo 3 el nuevo modelo asíncrono de gestión de tareas, y descritas en la sección anterior las características de OpenCL más relevantes para este trabajo, se presentará la implementación del mismo a la librería Controllers, explicando además la reestructuración de código realizada para poder incorporar a la misma la nueva versión del modelo.

Es importante recordar que el anterior modelo asíncrono estaba dirigido exclusivamente a su explotación en GPUs de NVIDIA, por lo que esa parte del código de la librería estaba fuertemente acoplada con CUDA, llamando directamente a funciones de la interfaz de programación de dicha arquitectura. Para incorporar esta nueva versión basada en OpenCL, en cambio, es fundamental desacoplar por completo *Controllers* y la API de CUDA, ya que habrá plataformas heterogéneas donde no haya una implementación de CUDA disponible contra la que compilar la librería.

Se decidió así dividir el proyecto en dos módulos independientes: un módulo *Kernel* que se encarga de la reescritura del código de usuario convirtiéndolo en código final propio de cada arquitectura, y un módulo *Core* encargado de la gestión de la cola de tareas y el lanzamiento de las mismas. Para desacoplar completamente el uso de CUDA y poder incorporar además el modelo implementado en OpenCL, ambos módulos principales contienen submódulos con las implementaciones propias de cada arquitectura soportada, incluyendo únicamente aquellos que se hayan especificado en la configuración del proyecto. Esta compilación condicional en base a la configuración del proyecto se consigue gracias al uso de CMake, que permite al usuario tanto elegir explícitamente mediante opciones de configuración qué implementaciones desea añadir, como

dejar que sea la propia herramienta CMake la que lo decida en función de la disponibilidad de cada plataforma.

4.2.1 *Módulo Kernel*

El módulo Kernel se encarga de transformar el código de usuario en código ejecutable por un dispositivo concreto, transformando adecuadamente los tipos, roles y *kernels* definidos por el programador, ya fuesen estos genéricos o dirigidos a una arquitectura concreta:

En el caso de OpenCL, esta reescritura involucra conceptos como:

- Los argumentos del kernel (tiles) y su tipo de acceso (lectura o lectura/escritura).
- El ajuste del número de hilos de los espacios de trabajo globales y locales, junto con el código necesario para evitar que los kernels excedan al ejecutarse las dimensiones fijadas.
- El cambio de índices para conseguir coalescencia en el acceso a memoria.
- El envío de *tiles* al kernel por partes, y su posterior reconstrucción interna.
- La creación, inicialización y destrucción de todas las estructuras propias de OpenCL específicas de cada kernel, como la estructura *program* y la estructura *kernel* .

La ejecución de los kernels (tareas de computación) necesita de información específica de cada arquitectura, detallada más adelante, que tiene que ser provista por el controlador específico de dicha arquitectura. Por ello, al lanzamiento de kernel se le ha añadido un campo de tipo unión para que cada arquitectura utilice su versión propia de dicho unión con toda la información necesaria. De esta forma se consigue enviar la información necesaria para el lanzamiento real de un kernel desde el controlador, soportando así la necesaria separación entre arquitecturas en la nueva implementación de la librería.

4.2.2 *Módulo Core*

El módulo Core se encarga de la gestión de la cola de tareas del controlador, así como de la ejecución de dichas tareas. Contiene una serie de controladores específicos de las diferentes arquitecturas fijadas en la configuración del proyecto, junto con una parte común encargada de encolar y desencolar tareas. Estas tareas contienen toda la información necesaria para que un controlador concreto pueda ejecutarlas.

Para que esto sea posible, ya que las tareas requieren de información específica de la arquitectura sobre la que se van a ejecutar, se añade un campo de tipo unión a los tiles que participan en la tarea, siendo este campo gestionado por cada controlador concreto en base a su arquitectura. Esos campos contienen toda la información necesaria sobre su estado interno, incluida la información

necesaria para posibles solapamientos y la gestión de dependencias. Este campo se trata en detalle más adelante.

4.3 IMPLEMENTACIÓN DEL BACKEND OPENCL

Una vez reestructurada la librería, y habiendo aislando todo lo relativo a CUDA en los submódulos correspondientes de forma que éstos solamente se incorporen de forma condicional al proyecto si así se indica en tiempo de compilación, se añade la implementación o *backend* de OpenCL. Así, en esta sección primero se definen las estructuras de datos propias de *Controllers* necesarias para el funcionamiento del modelo en OpenCL. Por último se detalla el funcionamiento de la implementación de OpenCL para los modelos de ejecución síncrona y asíncrona.

4.3.1 *Campos y estructuras específicas para OpenCL*

El controlador debe contener aquellas estructuras de OpenCL necesarias para el funcionamiento del backend, habiendo incorporado al mismo los siguientes campos:

- *policy*: Política de ejecución (síncrona o asíncrona) escogida por el usuario.
- *p_queue*: Referencia a la cola de tareas de *Controllers*.
- *platform*: Plataforma OpenCL escogida por el usuario.
- *device*: Dispositivo OpenCL escogido por el usuario para la ejecución de tareas.
- *context*: Contexto de OpenCL asociado a dicho dispositivo.
- *main_command_queue*: Cola principal. En la política síncrona se utiliza para todas las tareas (transferencia de datos, computación y sincronización), mientras que en la asíncrona se dedica exclusivamente a las tareas de computación.
- *read_command_queue*: Cola propia de la política asíncrona dedicada a las tareas de transferencia de datos del dispositivo al host.
- *write_command_queue*: Cola propia de la política asíncrona dedicada a las tareas de transferencia de datos del host al dispositivo.
- *last_kernel_event*: Evento asociado al último comando de lanzamiento de kernel (tarea de computación) ejecutado.

Así mismo, es necesario incorporar una estructura adicional en los tiles para guardar su información propia. El tile cuenta con un campo genérico en el que cada implementación guarda una estructura con los datos que necesite. La versión que el backend de OpenCL mantiene para dicha estructura se compone de los siguientes campos:

- *p_ctrl*: Referencia al controlador que va a utilizar el tile, de modo que el propio tile pueda acceder a los atributos o campos del controlador.
- *is_internal*: Campo de tipo lógico (boolean) que indica si la memoria (el buffer) es accesible desde el host o existe solo en el dispositivo. Esta variable de control es necesaria debido a que no es posible realizar tareas de transferencia de datos sobre variables internas del dispositivo.
- *is_pinned*: Campo de tipo lógico (boolean) que indica si la memoria del tile es de tipo *pinned* o no. Este tipo de memoria siempre está asociada a un buffer en el host, por lo que un tile de tipo interno esta variable de control siempre será falsa.
- *p_mem*: Referencia al buffer de memoria que representa al tile en el dispositivo. Puede no existir si el tile todavía no ha sido vinculado con un dispositivo.
- *offloading_event_read*: Evento para las tareas de transferencia de datos del dispositivo al host.
- *offloading_event_write*: Evento para las tareas de transferencia de datos del host al dispositivo.
- *kernel_event_read*: Evento para las tareas de computación que utilicen el tile como entrada (IN) o entrada / salida (IO).
- *kernel_event_write*: Evento para las tareas de computación que utilicen el tile como salida (OUT) o entrada / salida (IO).

Por otra parte, la ejecución real de una tarea de computación ocurre fuera del ámbito del controlador de OpenCL, en el módulo Kernel. Para llevar a cabo esa ejecución el controlador de OpenCL debe proveer de los tiles y las estructuras de OpenCL necesarias. Estas estructuras se encapsulan en otra de tipo unión llamada *request*, que el controlador rellena y envía al módulo correspondiente. En el caso de OpenCL, una *request* contiene:

- *p_queue*: Referencia a la cola de ejecución (*main_command_queue*) a la que enviar el kernel.
- *p_task_arguments*: Lista de tiles participantes en la computación, respetando el orden que éstos siguen en la definición del kernel.
- *p_roles*: Lista de los roles de los tiles participantes en la computación (IN, OUT, IO), ordenados.
- *p_event_wait_list*: Lista de eventos por los que la tarea de computación tiene que esperar.
- *p_kernel_event*: Referencia al evento que generará la tarea de computación, y que será usado por los tiles asociados para actualizar sus eventos en base al rol que tienen en la tarea (IN, OUT, IO).

4.3.2 *Funcionamiento general del controlador de OpenCL*

En el capítulo anterior se introdujeron de forma general los conceptos de *hilos de ejecución* en Controllers, haciendo distinción entre *hilo del host* e *hilo del controlador*. A continuación se describen las responsabilidades específicas de ambos hilos en la implementación de OpenCL.

Por una parte, el hilo del controlador se responsabiliza de las siguientes funciones:

- *Inicialización del controlador de OpenCL*: En esta fase se crean e inicializan las estructuras de datos descritas en el apartado inmediatamente anterior. El controlador recibe la política, la plataforma y el dispositivo del usuario y en base a ello inicializa las estructuras de OpenCL necesarias (plataforma, dispositivo, contexto, colas de comandos, etc). Es en este punto donde se crea el hilo propio del controlador.

- *Encolado de tareas*: El hilo del host es el encargado, mediante la interfaz que proporciona *Controllers*, de añadir tareas en la cola propia de la librería. Al añadir una tarea que requiera de sincronización, el hilo del host se bloqueará hasta que le despierte el hilo del controlador. Se detalla este proceso más adelante.

- *Destrucción del controlador*: El hilo del host inicia el proceso de destrucción del controlador, que procede a liberar sus recursos y terminar su ejecución. El hilo del host se bloquea hasta que el controlador termine este proceso de destrucción y liberación de recursos, finalizando con la destrucción del hilo propio del controlador.

Por otra, el hilo del controlador se centra en ejecutar el bucle principal de ejecución y extracción de tareas de la cola. Una vez inicializado el controlador, y creado el hilo de ejecución del mismo, este hilo ejecuta en un bucle infinito que permanentemente intenta desencolar tareas de la cola del controlador. En cada iteración comprueba si hay tareas en la cola. Si encuentra tareas, desencola la primera y la evalúa en función de la política síncrona o asíncrona previamente establecida. Este comportamiento se mantiene hasta que se lance el proceso de destrucción del hilo, momento en que termina el bucle y libera las estructuras de datos inicializadas en la creación del controlador. Una vez terminada la ejecución del hilo del controlador, éste es finalmente destruido.

4.3.3 *Creación y evaluación síncrona de tareas*

A continuación se describe el comportamiento específico de los hilos de host y de controlador para una política síncrona de gestión de tareas.

En cuanto al hilo del host, éste se comporta siempre de la misma manera en modo síncrono: crea una nueva tarea, la añade a la cola de tareas del controlador y se bloquea a la espera de que la ejecución de dicha tarea termine. La excepción son las tareas de computación (lanzamiento de kernels), que como ya se ha comentado siempre son asíncronas en OpenCL. Cualquier tarea posterior con la política de ejecución síncrona funciona, por tanto, como un punto de sincronización. Las tareas de sincronización *waitTile* y *globalSync* pueden ser usadas en la política síncrona, si bien no son necesarias. La sobrecarga que éstas suponen es despreciable, y permiten que el mismo código pueda ser luego ejecutado siguiendo una política asíncrona sin modificaciones adicionales. Además, explicitar siempre en el código los puntos de sincronización (usando en este caso las funciones provistas por *Controllers*), está considerado como una buena práctica de programación de aplicaciones paralelas.

El hilo del controlador, por su parte, evalúa cada tarea y la ejecuta. Al haber fijado una política síncrona, obvia completamente las dependencias entre tareas. Las tareas de transferencia de datos (*moveFrom* y *moveTo*) y las de computación (*launchKernel*) se ejecutan como comandos en la cola principal (main) de OpenCL del controlador. El controlador realiza una espera mediante la función *clWaitForEvents* por cada comando que se introduce en la cola de comandos de OpenCL, con la excepción de las tareas de computación, por lo que el hilo del controlador no continua su ejecución hasta que el comando en cuestión finalice. Las tareas de gestión de memoria (*attach*, *createInternal*, *dettach* y *destroyInternal*) se ejecutan con las funciones correspondientes de OpenCL (como *clCreateBuffer* y *clReleaseMemObject*) en modo bloqueante, por lo que el hilo del controlador espera a que éstas finalicen. Las tareas de sincronización (*waitTile* y *globalSync*) se ejecutan invocando las funciones de OpenCL *clWaitForEvents* y *clFinish* sobre la cola de comandos principal. Estas funciones realizan una espera por la ejecución de los comandos asociados a la lista de eventos en el primer caso, y por la ejecución de todos los comandos de la cola en el segundo. Una vez terminada la espera el hilo de controladores continua su ejecución. Al final de cada tarea en el modo síncrono, con la excepción antes comentada de las tareas de computación, el hilo del controlador se encarga de despertar el hilo del host, y continua con la evaluación de la siguiente tarea de la cola de Controladores.

4.3.4 *Creación y evaluación asíncrona de tareas*

La ejecución en la política asíncrona comparte muchos puntos con la síncrona. Así, la creación y evaluación de las tareas de gestión de memoria (*attach*, *createInternal*, *dettach* y *destroyInternal*) y la creación y posterior destrucción y liberación de recursos funcionan de igual manera, con los mismos bloqueos y esperas que la versión síncrona. Las tareas de sincronización (*waitTile* y *globalSync*) también funcionan de la misma forma en esta política, pero como se verá a lo largo de este apartado, ahora sí que son necesarias como puntos de sincronización.

Para conseguir solapar tareas en OpenCL, como se expone al principio del capítulo, es necesario el uso de distintas colas de comandos, tres en total. Una para las transferencias de datos del host al dispositivo, otra para las transferencias de datos del dispositivo al host y una última para las tareas de computación.

Las tareas de computación son asíncronas en ambas políticas, y se siguen ejecutando como un comando en la cola principal de OpenCL del controlador, aunque ahora se gestionarán efectivamente las posibles dependencias de acuerdo con lista de eventos de espera que recibe el comando de lanzamiento de kernel. Estos eventos están relacionados con los tiles que participan en la tarea en base a su rol (IN, OUT, o IO) específico en la misma. Una vez añadida la tarea de computación como comando en la cola principal, el evento que identifica ese comando se utiliza para sobrescribir los eventos correspondientes de cada tile participante en la tarea, de acuerdo al modelo expuesto en el capítulo anterior. Dado el carácter asíncrono de las tareas de lanzamiento de kernel en OpenCL, ni el hilo del host se bloquea al crearlas ni el hilo del controlador espera por la finalización de las mismas ni tampoco necesita despertar al host, puesto que no se bloquea.

La mayor diferencia con la política síncrona reside en la gestión de las tareas de transferencia de datos (*moveFrom* y *moveTo*). El hilo del host no se bloquea al crear y añadir estas tareas a la cola de tareas, continuando así su ejecución pudiendo añadir más tareas a dicha cola. Este comportamiendo es condición imprescindible para soportar el solapamiento de las mismas, ya que resulta evidente que no se puede solapar una única tarea.

El hilo del controlador, al evaluar una de estas tareas, genera una lista de eventos de espera en base a los eventos del tile y el tipo de tarea, lanza el comando en la cola de OpenCL correspondiente del controlador (*read_command_queue* para *moveFrom* y *write_command_queue* para *moveTo*) utilizando las funciones de OpenCL antes mencionadas (*clEnqueueReadBuffer* y *clEnqueueWriteBuffer*, respectivamente) y actualiza el evento correspondiente en el tile, como se expone en el capítulo anterior.

El hilo del controlador no espera por la ejecución de esos comandos, sino que continúa con su ejecución y la evaluación de la siguiente tarea. Además, como el hilo del host no está bloqueado, tampoco hay que despertarlo.

De esta forma, gracias al uso de tres colas de comandos de OpenCL diferentes y el uso de eventos para gestionar las dependencias entre tareas en colas de comandos diferentes, se consigue el solapamiento de tareas entre las tres colas. Esta implementación y este modelo permiten el solapamiento de hasta tres tareas al mismo tiempo, al permitir solapar la ejecución de las tres colas de comandos, siempre que el hardware subyacente y las dependencias entre las tareas de la aplicación paralela lo permitan. En el capítulo siguiente se validará experimentalmente este modelo mediante una batería de pruebas.

5

VALIDACIÓN EXPERIMENTAL DEL MODELO ASÍNCRONO BASADO EN EVENTOS

Este capítulo contiene:

- Definición de las estructuras y cargas de trabajo utilizadas en la bateria de pruebas.
- Batería de pruebas.
- Conclusiones.

La validación experimental del modelo se ha realizado mediante un conjunto de pruebas que ejecutan de forma síncrona y asíncrona operaciones implementadas usando primero código OpenCL nativo, y las funciones propias de *Controllers* después.

El objetivo de estas pruebas es demostrar que los resultados de ambas versiones coinciden, y que los tiempos se reducen en los casos con solapamiento. Dado que solo tiene sentido probar la corrección del solapamiento de tareas en modo asíncrono, en este capítulo se describirán exclusivamente las pruebas siguiendo dicha política de gestión de tareas.

5.1 DEFINICIÓN DE LAS ESTRUCTURAS Y CARGAS DE TRABAJO UTILIZADAS

Previamente a la descripción de la batería de pruebas realizadas, se definen los conjuntos de estructuras de datos (tiles) y de cargas de computación (kernels) definidas a tal efecto y utilizadas en las mismas.

5.1.1 *Estructuras de datos utilizadas*

Se utilizan dos tipos de tiles, *ShortTile*, con un vector de datos de 1000 elementos de tipo float, y *LongTile*, con un vector de datos de 10000000 elementos de tipo float. Los datos concretos con los que se inicializa cada tile se indican en la prueba correspondiente. Además, en aquellos casos en los que el tile esté involucrado en una tarea de computación (kernel), se indica su rol en la misma (IN, OUT o IO).

La definición de dos tamaños tan diferentes se justifica por la necesidad de disponer de cargas de trabajo lo suficientemente diferentes como para generar tareas de duraciones muy distintas, y así poder comprobar todos los casos posibles de ejecución de tareas en el modelo.

5.1.2 *Cargas de computación utilizadas*

De forma similar a los tiles, definimos dos tipos de kernels (tareas de computación) *ShortKernel* y *LongKernel*. Se han diseñado de forma que *LongKernel* tarde unas 1000 veces más tiempo en ejecutarse que *ShortKernel*. El resultado final de ambos kernels es copiar los valores del primer tile de entrada (IN o IO) en todos los tiles de salida (IO o OUT). Los argumentos de los kernels se definen de forma específica en cada una de las pruebas de la batería.

La corrección del modelo se demuestra comprobando los resultados de la ejecución de la secuencia de tareas de cada una de las pruebas en las cuatro implementaciones: OpenCL nativo síncrono, OpenCL nativo asíncrono, *Controllers* síncrono y *Controllers* asíncrono. Los resultados deben coincidir en todas las ejecuciones y, en los casos con solapamiento, las versiones asíncronas deben ejecutarse en menos tiempo.

5.1.3 *Tareas auxiliares para la definición y ejecución de las pruebas*

Son necesarias para la definición y ejecución de las pruebas otras dos tareas. La primera es *globalSync*, función provista por *Controllers* para sincronizar por completo el dispositivo y que no recibe tiles como argumentos, sino que simplemente espera por la finalización de todas las tareas previas. La segunda, denominada *changeTile(tile, valor)*, fija el valor indicado como argumento en todos los elementos de un tile en el host.

5.2 BATERÍA DE PRUEBAS

Se consideran correctas por ensayo experimental cada una de las tareas que define Controladores por separado, ante la imposibilidad de aislar una operación al ejecutarse sobre un dispositivo (co-procesador) y no sobre el host. Todas las pruebas, por lo tanto, tienen al menos una tarea de *moveTo* y al menos una tarea de *moveFrom*.

Cada prueba se define como una colección de tiles y una secuencia de tareas. Cada tarea de la secuencia indica los tiles que utiliza y, si es aplicable, el rol de los mismos en la tarea.

5.2.1 *T-01*

Nombre: T-01.
Descripción: Dependencia entre tareas de transferencia de datos *moveTo* y *moveFrom*.
Tiles utilizados:

- *LongTile long_tile_a*

Kernels utilizados: Ninguno.
Secuencia de tareas:

1. *changeTile(long_tile_a, 0)*
2. *moveTo(long_tile_a)*
3. *globalSync()*
4. *changeTile(long_tile_a, 1)*
5. *globalSync()*
6. *moveTo(long_tile_a)*
7. *moveFrom(long_tile_a)*
8. *globalSync()*

Se espera solapamiento: No.
Resultado esperado: *long_tile_a* con el valor 1 en todos sus elementos.
Se consigue solapamiento: No.
Resultado: *long_tile_a* con el valor 1 en todos sus elementos.

5.2.2 *T-02*

Nombre: T-02.
Descripción: Solapamiento entre tareas de transferencia de datos *moveTo* y *moveFrom*.
Tiles utilizados:

- *LongTile long_tile_a*

- *LongTile long_tile_b*

Kernels utilizados: Ninguno.
Secuencia de tareas:

1. *changeTile(long_tile_a, 1)*
2. *changeTile(long_tile_b, 2)*
3. *moveTo(long_tile_a)*
4. *globalSync()*
5. *changeTile(long_tile_a, 0)*
6. *globalSync()*
7. *moveTo(long_tile_b)*
8. *moveFrom(long_tile_a)*
9. *globalSync()*

Se espera solapamiento: Sí.
Resultado esperado: *long_tile_a* con el valor 1 en todos sus elementos.
Se consigue solapamiento: Sí.
Resultado: *long_tile_a* con el valor 1 en todos sus elementos.

5.2.3 *T-03*

Nombre: T-03.
Descripción: Dependencia entre una tarea de computación *longKernel* y dos tareas de transferencia de datos *moveTo* y *moveFrom*.
Tiles utilizados:

- *LongTile long_tile_a*
- *LongTile long_tile_b*

Kernels utilizados:

- *LongKernel long_kernel(IN long_tile_a, OUT long_tile_b)*

Secuencia de tareas:

1. *changeTile(long_tile_a, 1)*
2. *changeTile(long_tile_b, 0)*
3. *globalSync()*
4. *moveTo(long_tile_a)*
5. *kernelLaunch(long_kernel)*
6. *moveFrom(long_tile_b)*
7. *globalSync()*

Se espera solapamiento: No.
Resultado esperado: *long_tile_b* con el valor 1 en todos sus elementos.
Se consigue solapamiento: No.

Resultado: *long_tile_b* con el valor 1 en todos sus elementos.

5.2.4 *T-04*

Nombre: T-04.
Descripción: Solapamiento entre una tarea de computación *longKernel* y una tarea de transferencia de datos *moveTo*.
Tiles utilizados:

- *LongTile long_tile_a*
- *LongTile long_tile_b*
- *LongTile long_tile_c*

Kernels utilizados:

- *LongKernel long_kernel(IN long_tile_a, OUT long_tile_b)*

Secuencia de tareas:

1. *changeTile(long_tile_a, 1)*
2. *changeTile(long_tile_b, 0)*
3. *changeTile(long_tile_c, 2)*
4. *moveTo(long_tile_a)*
5. *globalSync()*
6. *kernelLaunch(long_kernel)*
7. *moveTo(long_tile_c)*
8. *moveFrom(long_tile_b)*
9. *globalSync()*

Se espera solapamiento: Sí.
Resultado esperado: *long_tile_b* con el valor 1 en todos sus elementos.
Se consigue solapamiento: Sí.
Resultado: *long_tile_b* con el valor 1 en todos sus elementos.

5.2.5 *T-05*

Nombre: T-05.
Descripción: Solapamiento entre una tarea de computación *longKernel* y una tarea de transferencia de datos *moveFrom*.
Tiles utilizados:

- *LongTile long_tile_a*
- *LongTile long_tile_b*
- *LongTile long_tile_c*

Kernels utilizados:

- *LongKernel long_kernel(IN long_tile_a, OUT long_tile_b)*

Secuencia de tareas:

1. *changeTile(long_tile_a, 1)*
2. *changeTile(long_tile_b, 0)*
3. *changeTile(long_tile_c, 2)*
4. *moveTo(long_tile_a)*
5. *moveTo(long_tile_c)*
6. *changeTile(long_tile_c, 0)*
7. *globalSync()*
8. *kernelLaunch(long_kernel)*
9. *moveFrom(long_tile_c)*
10. *moveFrom(long_tile_b)*
11. *globalSync()*

Se espera solapamiento: Sí.
Resultado esperado: *long_tile_b* con el valor 1 en todos sus elementos. *long_tile_c* con el valor 2 en todos sus elementos.
Se consigue solapamiento: Sí.
Resultado: *long_tile_b* con el valor 1 en todos sus elementos. *long_tile_c* con el valor 2 en todos sus elementos.

5.2.6 *T-06*

Nombre: T-06.
Descripción: Solapamiento entre una tarea de computación *longKernel* y dos tareas de transferencia de datos *moveTo* y *moveFrom*.
Tiles utilizados:

- *LongTile long_tile_a*
- *LongTile long_tile_b*
- *LongTile long_tile_c*
- *LongTile long_tile_d*

Kernels utilizados:

- *LongKernel long_kernel(IN long_tile_a, OUT long_tile_b)*

Secuencia de tareas:

1. *changeTile(long_tile_a, 1)*
2. *changeTile(long_tile_b, 0)*
3. *changeTile(long_tile_c, 2)*
4. *changeTile(long_tile_d, 3)*
5. *moveTo(long_tile_a)*

6. *moveTo(long_tile_c)*
7. *changeTile(long_tile_c, 0)*
8. *globalSync()*
9. *kernelLaunch(long_kernel)*
10. *moveFrom(long_tile_c)*
11. *moveTo(long_tile_d)*
12. *moveFrom(long_tile_b)*
13. *globalSync()*

Se espera solapamiento: Sí.
Resultado esperado: *long_tile_b* con el valor 1 en todos sus elementos. *long_tile_c* con el valor 2 en todos sus elementos.
Se consigue solapamiento: Sí.
Resultado: *long_tile_b* con el valor 1 en todos sus elementos. *long_tile_c* con el valor 2 en todos sus elementos.

5.2.7 *T-07*

Nombre: T-07.
Descripción: Dependencia entre una tarea de computación *shotKernel* y dos tareas de transferencia de datos *moveTo* y *moveFrom*. Se busca que la ejecución de la transferencia de datos dure menos que la tarea de la computación, de forma que la tarea *moveTo* espere a la finalización del kernel.
Tiles utilizados:

- *LongTile short_tile_a*
- *LongTile short_tile_b*

Kernels utilizados:

- *LongKernel long_kernel(IN short_tile_a, OUT short_tile_b)*

Secuencia de tareas:

1. *changeTile(short_tile_a, 1)*
2. *changeTile(short_tile_b, 0)*
3. *moveTo(short_tile_a)*
4. *globalSync()*
5. *changeTile(short_tile_a, 2)*
6. *globalSync()*
7. *kernelLaunch(long_kernel)*
8. *moveTo(short_tile_a)*
9. *moveFrom(short_tile_b)*
10. *moveFrom(short_tile_a)*
11. *globalSync()*

Se espera solapamiento: Sí.
Resultado esperado: *short_tile_a* con el valor 2 en todos sus elementos. *short_tile_b* con el valor 1 en todos sus elementos.
Se consigue solapamiento: Sí.
Resultado: *short_tile_a* con el valor 2 en todos sus elementos. *short_tile_b* con el valor 1 en todos sus elementos.

5.2.8 *T-08*

Nombre: T-08.
Descripción: Dependencia entre una tarea de computación *shortKernel* y dos tareas de transferencia de datos *moveTo* y *moveFrom*. Se busca que la ejecución de la transferencia de datos dure mas que la tarea de la computación, de forma que la tarea *kernelLaunch* espere a la finalización de la transferencia.
Tiles utilizados:

- *LongTile long_tile_a*
- *LongTile long_tile_b*

Kernels utilizados:

- *shortKernel short_kernel(IN long_tile_a, OUT long_tile_b)*

Secuencia de tareas:

1. *changeTile(long_tile_a, 1)*
2. *changeTile(long_tile_b, 0)*
3. *moveTo(long_tile_a)*
4. *globalSync()*
5. *changeTile(long_tile_a, 2)*
6. *globalSync()*
7. *moveTo(long_tile_a)*
8. *kernelLaunch(short_kernel)*
9. *moveFrom(long_tile_a)*
10. *moveFrom(long_tile_b)*
11. *globalSync()*

Se espera solapamiento: Sí.
Resultado esperado: *long_tile_a* con el valor 2 en todos sus elementos. *long_tile_b* con el valor 2 en todos sus elementos.
Se consigue solapamiento: Sí.
Resultado: *long_tile_a* con el valor 2 en todos sus elementos. *long_tile_b* con el valor 2 en todos sus elementos.

5.3 CONCLUSIONES

Tras la ejecución de la batería de pruebas, y habiendo obtenido resultados satisfactorios en todas ellas, se considera válido el modelo de ejecución asíncrona propuesto, y correcta la implementación en OpenCL del mismo.

El modelo y su implementación, por lo tanto proporcionan un mecanismo validado para la programación de aplicaciones paralelas cuyas dependencias permitan el solapamiento de tareas de computación y transferencia de datos, lo que permite mejorar los tiempos de ejecución de estas. Además el modelo permite al usuario ignorar la gestión de dependencias, las cuales se ha demostrado que son gestionadas internamente de forma correcta.

6

ESTUDIO EXPERIMENTAL

Este capítulo contiene:

- Descripción y objetivos del estudio y sus casos.
- Estudio del rendimiento de los casos propuestos.
- Métricas de esfuerzo de desarrollo.
- Conclusiones.

6.1 DESCRIPCIÓN, OBJETIVOS Y CASOS DE ESTUDIO

En esta sección se presentan los objetivos de este estudio experimental, el cual pretende demostrar las ventajas que proporciona el uso de la librería *Controllers* para el desarrollo de aplicaciones paralelas sobre sistemas heterogéneos.

Se han implementado códigos en versiones síncrona y asíncrona, en combinación tanto con las librerías de referencia (CUDA, OpenCL) como con *Controllers*. Todos los ejemplos han sido desarrollados usando C99 y compilados con GCC v7.2.

6.1.1 *Objetivos*

A continuación se enumeran los principales objetivos del presente estudio experimental:

1. Comprobar la mejora de rendimiento que aporta el solapamiento de tareas.
2. Demostrar que la magnitud de la mejora de rendimiento obtenida gracias al solapamiento de tareas depende de las propiedades de la aplicación desarrollada y de la relación de carga de trabajo entre las transferencias de datos y de computación de la misma.

3. Demostrar que el uso de la librería *Controllers* para el desarrollo de aplicaciones paralelas reduce el esfuerzo de desarrollo por parte del usuario para la programación de las mismas.
4. Calcular la sobrecarga que introduce la librería *Controllers* respecto de la gestión de tareas y dependencias.

6.1.2 *Descripción de los sistemas sobre los que se ejecuta la experimentación*

Se han utilizado dos sistemas diferentes, con el objeto de demostrar las posibilidades de ejecución de la librería en entornos heterogéneos. A continuación se recogen las principales características de ambos:

- **Plataforma AMD**:
  - Procesador Intel i5-6600 @ 3.9 GHz.
  - Memoria RAM 16GB GV GDDR3.
  - Co-procesador (GPU) AMD R9 380X (2048 cores @ 1030 MHz, 4GB GDDR5).
  - Sistema Operativo Ubuntu 16.04
  - Driver amdgpu-pro con OpenCL 1.2.

- **Plataforma NVIDIA**:
  - Procesador i5-3330 @ 3.0 GHz,
  - Memoria RAM 8GB GV GDDR3.
  - Co-procesador (GPU) NVIDIA Tesla K40c (2880 cores @ 745 MHz, 12GB GDDR4).
  - Sistema Operativo CentOS 7.
  - Driver CUDA 9.0 con OpenCL 1.2.

6.1.3 *Casos de estudio*

Se han elegido 3 aplicaciones diferentes como casos de estudio. Cada uno de ellos tiene unas características concretas en relación a las cargas de computación y transferencia de datos, de manera que el análisis conjunto de todos ellos permite observar el comportamiento en un amplio espectro de relaciones de carga entre tareas de computación y de transferencia de datos.

6.1.4 *Stencil Computation*

Este algoritmo calcula el punto de estabilidad de la ecuación diferencial parcial de Poisson para la difusión del calor, usando el método de Jacobi iterativo sobre una matriz bidimensional. Es de tipo stencil de 4 puntos, y se ejecuta un numero fijo de veces (iteraciones). En cada iteración

calcula un nuevo valor para cada una de las celdas de la matriz, usando las celdas vecinas.

Se han elegido 100 iteraciones para el estudio, con tamaños de matriz entre 10000x10000 y 20000x20000. Cada 10 iteraciones se transfiere el resultado al host y se copia el resultado a un vector auxiliar, que podría ser usado por el host para comprobar resultados parciales, realizar cálculos adicionales, etc. De esta manera se simulan situaciones en las que es posible seguir ejecutando tareas en el dispositivo a la vez que el host realiza trabajo con resultados anteriores.

Este caso de estudio tiene una carga pequeña de computación en comparación con la carga de las tareas de transferencia de datos.

6.1.5 *Potencia de matrices (Matrix Pow)*

En este caso se calcula iterativamente la multiplicación de una matriz, a la que llamaremos inicial, por el resultado de multiplicar el resultado de cada iteración por dicha matriz inicial, y comenzando con la multiplicación de la matriz inicial por si misma. Al final de cada iteración se copia el resultado parcial calculado en el dispositivo de vuelta al host, utilizándolo para calcular una norma especial en los resultados parciales que se guarda en un vector. Nótese que estos cálculos de la norma en el host se van solapando de forma asíncrona con la ejecución en el dispositivo de la siguiente iteración del algoritmo.

Se ha elegido un número fijo de 10 iteraciones, y unos tamaños de matrices cuadradas entre 1024x1024 y 5120x5120. En este caso de estudio la carga de trabajo de las tareas de computación es notablemente más grande que la carga que suponen las tareas de transferencia de datos.

Es importante recordar que la multiplicación de matrices es una de las operaciones de cómputo más costosas y, por tanto, más estudiadas en el ámbito de la programación paralela. Así, este algoritmo está adaptado a partir de los ejemplos que provee CUDA en sus herramientas. Dicho ejemplo está altamente optimizado y requiere matrices cuadradas para su ejecución.

6.1.6 *Filtro de Sobel (Sobel filter)*

El filtro de Sobel es un *kernel* de computación muy utilizado para la detección de bordes en imágenes en escala de grises. Aplica dos operaciones de tipo *stencil* a la imagen de entrada, aproximando así las derivadas en los ejes X e Y, y a continuación calcula el gradiente de cada celda para las matrices de derivadas. La imagen generada a partir de la matriz que contiene el gradiente

calculado para cada celda es el resultado de aplicar dicho filtro.

Para cada imagen, la operación solamente necesita una iteración para completarse. El caso de estudio seleccionado representa la aplicación de este filtro a un fragmento de video compuesto de 100 fotogramas, de modo que realiza un total de 100 iteraciones. Se han ejecutado pruebas con los tamaños de matrices correspondientes a resoluciones de vídeo ultra-HD (4K, 8K 16K). En este caso, cada iteración supone enviar un fotograma del host al dispositivo, que éste ejecute la computación del filtrado, y que el host reciba como resultado la matriz de gradientes correspondiente al fotograma de entrada.

Para este caso de estudio se puede considerar que la carga de trabajo de las tareas de computación y de transferencia de datos se encuentra equilibrada. Así mismo, en la figura 7 se pueden ver la evolución temporal de las tareas ejecutadas por *Controllers* para este caso de estudio tanto en modo síncrono como asíncrono. Nótese en la línea de tiempo inferior cómo se produce el solapamiento de las tareas de transferencia y de computación cuando se sigue una política asíncrona.

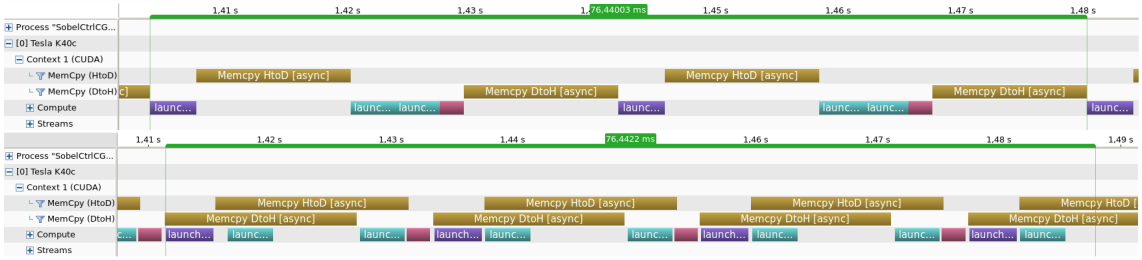

Figura 7: Líneas de tiempo generadas por el programa CUDA 9.0 Visual Profiler en las que se muestran 76 ms de la ejecución de un filtro de Sobel implementado con *Controllers*. El modo síncrono se encuentra en la parte superior de la figura, y el asíncrono en la parte inferior.

6.2 EVALUACIÓN DEL RENDIMIENTO

En esta sección se muestran los resultados de la experimentación realizada en términos de rendimiento computacional para los casos de prueba *Stencil Computation* y *Matrix Pow*. En la figura 8 se pueden ver las gráficas de rendimiento para las diferentes ejecuciones de las distintas versiones de dichos casos en las 2 máquinas antes descritas. El eje horizontal indica los tamaños de entrada, mientras que el vertical indica el tiempo de ejecución en segundos.

Para ambos ejemplos se puede observar que las curvas de rendimiento obtenidas para las versiones nativas y las implementadas con *Controladores*, agrupadas según la política de ejecución elegida, son muy similares. Esto demuestra que la sobrecarga añadida por la librería frente a las versiones nativas es despreciable en ejecuciones con una carga de trabajo lo suficientemente importante. Por otra parte, la diferencia entre los pares de gráficas asociados a la política asíncrona y los asociados a la política síncrona demuestran la ganancia de rendimiento obtenida por las ejecuciones asíncronas con solapamiento frente a las síncronas. Las aceleraciones obtenidas en cada caso se muestran en forma porcentual, como etiquetas sobre las propias gráficas.

La tabla 1 detalla los tiempos de ejecución obtenidos para cada versión y política de gestión de tareas en el ejemplo del filtro de Sobel, junto con la aceleración conseguida por la ejecución asíncrona con solapamiento de tareas respecto de la síncrona. Los resultados de este ejemplo demuestran que un equilibrio entre entre las cargas solapables de computación y transferencia maximiza la mejora de rendimiento obtenida por el modelo de ejecución asíncrono implementado en este trabajo.

Sobel filter (CUDA-Backend on NVIDIA GPU)

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

Sobel filter (OpenCL-Backend on NVIDIA GPU)

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

Sobel filter (OpenCL-Backend on AMD GPU)

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

Tabla 1: Tiempos de ejecución obtenidos por *Sobel filter*. Los porcentajes de mejora de rendimiento del modo asíncrono respecto del síncrono se muestran en la úlitma columna.

6.3 MÉTRICAS DE ESFUERZO DE DESARROLLO

En esta sección se analiza y compara el esfuerzo de programación necesario para desarrollar las versiones síncronas y asíncronas de los casos de estudio *Stencil*, *Matrix Pow* y *Sobel filter* usando tanto la librería *Controllers* como las librerías nativas de referencia (CUDA, OpenCL).

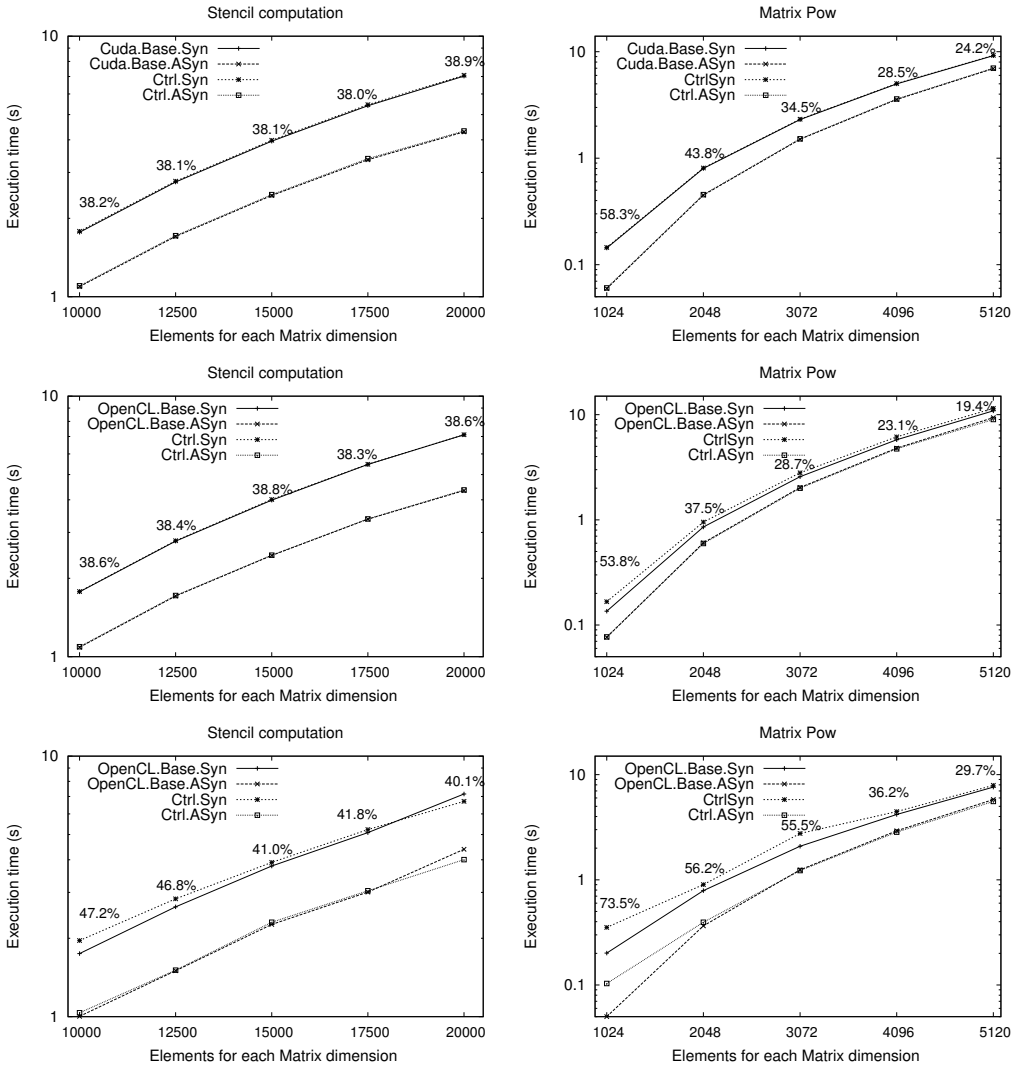

Figura 8: Tiempos de ejecución (segundos) de los programas base y de *Controllers* para los casos de estudio de *stencil* (izquierda) y potencia de una matriz (derecha). Primera fila: Programas base en CUDA, y back-end CUDA de Controller en plataforma NVIDIA. Segunda fila: Programas base en OpenCL, y backend OpenCL de Controller en plataforma NVIDIA. Tercera fila: Programas base en OpenCL, y backend OpenCL de Controller en plataforma AMD.

Se ha evaluado dicho esfuerzo mediante cuatro métricas diferentes: número de lineas de código, número de tokens, complejidad ciclomática de McCabe [12] y esfuerzo de desarrollo de Halstead [5]. Las dos primeras miden el volumen del código que el programador tiene que escribir para cada caso. La complejidad ciclomática mide las divergencias que se producen a lo largo de la ejecución del código, siendo por tanto un indicador de la dificultad de depuración del mismo (a mayor complejidad ciclomática, mayor posibilidad de que haya errores lógicos en el código). La última métrica utiliza tanto la complejidad del código como el volumen del mismo para obtener una valoración global del esfuerzo de desarrollo.

Caso de estudio	Versión	LOC	TOK	CCN	Halst.
Stencil	Ctrl	86	605	8	15 390
	CUDA_Syn	96	715	14	19 540
	CUDA_Asyn	116	849	14	22 853
	OpenCL_Syn	135	1 272	13	28 585
	OpenCL_Asyn	180	1 479	14	33 481
Matrix Pow	Ctrl	133	645	8	12 953
	CUDA_Syn	141	1 022	22	21 506
	CUDA_Asyn	170	1 202	22	28 847
	OpenCL_Syn	179	1 650	18	29 553
	OpenCL_Asyn	234	1 950	21	33 715
Sobel filter	Ctrl	115	799	13	16 605
	CUDA_Syn	132	1 229	16	30 044
	CUDA_Asyn	144	1 270	16	32 814
	OpenCL_Syn	266	2 423	22	33 645
	OpenCL_Asyn	312	2 554	22	34 879

Tabla 2: Métricas de esfuerzo de desarrollo sobre los códigos implementados con *Controllers* (Ctrl) y las diferentes versiones de referencia. Estas métricas incluyen una comparación del número de lineas (LOC), número de *tokens* (TOK), complejidad ciclomática de McCabe (CCN) y la métrica de esfuerzo de desarrollo de Halstead (Halst.).

Como se puede ver en la tabla 2, el uso de *Controllers* implica una disminución importante del esfuerzo necesario para desarrollar aplicaciones paralelas. Destacan especialmente casos como la comparativa con la versión asíncrona en OpenCL del filtro de Sobel, cuyo desarrollo necesita alrededor del triple de esfuerzo de acuerdo con las distintas métricas obtenidas. Usando CUDA la diferencia se reduce ligeramente para la complejidad ciclomática y las métricas de volumen de código, si bien para la métrica de Halstead puede verse que el esfuerzo de desarrollo sigue siendo casi el doble respecto de la versión basada en *Controllers*. Casos como estos demuestran las ventajas derivadas de usar la librería enriquecida con el modelo asíncrono propuesto, cumpliendo el objetivo fijado de ofrecer al usuario un mecanismo de programación que le suponga un menor esfuerzo de desarrollo.

A modo ilustrativo, en las figuras 9 y 10 se muestran dos ejemplos de código del filtro de Sobel, uno implementado usando *Controllers* y otro usando la interfaz de programación de OpenCL nativo, respectivamente. A simple vista puede apreciarse la diferencia en volumen y claridad de código, ya que en la misma cantidad de líneas la versión de *Controladores* implementa la práctica totalidad del código principal del programa, mientras que en la versión de OpenCL solo el código del bucle principal de ejecución ya ocupa un número de lineas similar.

6.4 CONCLUSIONES

En esta sección se revisan los objetivos propuestos del estudio y se exponen las conclusiones derivadas del mismo, con respecto tanto a la mejora del rendimiento computacional como a la reducción del esfuerzo de desarrollo.

6.4.1 *Revisión de objetivos*

1. Se demuestra la mejora de rendimiento que aporta el solapamiento de tareas.
2. Se demuestra que la mejora de rendimiento conseguida gracias al solapamiento de tareas depende de la aplicación desarrollada y de la diferencia de carga entre transferencia de datos y computación de la misma.
3. Se demuestra que el uso de la librería *Controllers* para el desarrollo de aplicaciones paralelas sobre co-procesadores de fabricantes diferentes reduce el esfuerzo de desarrollo necesario en todos los casos.
4. Se demuestra que la sobrecarga introducida por *Controllers* debido a la gestión de tareas y dependencias es constante e independiente de los tamaños de entrada y del número de iteraciones, y además resulta despreciable en términos de tiempos de ejecución para aplicaciones con una carga computacional suficiente.

6.4.2 *Conclusiones*

La ejecución asíncrona permite que se realicen simultáneamente cálculos en el host y en el dispositivo, así como el solapamiento de tareas de transferencia de datos y computación. Ambas propiedades del modelo demuestran su utilidad a la hora de conseguir mejoras de rendimiento como las vistas en la experimentación.

Gracias a dicha política de gestión de tareas, la sobrecarga introducida por *Controllers* queda oculta por la ejecución de tareas en el dispositivo. Además, esta sobrecarga tiende ser constante, de modo que cuanto mayor sea la carga de computación de una aplicación en si, menor será el

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

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

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

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

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

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

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

Figura 9: Ejemplo de código del filtro de Sobel implementado usando *Controllers* en modo asíncrono. Se muestra la inicialización de tiles, el bucle principal de ejecución, la liberación de recursos y la destrucción del controlador.

```
1  int main(int argc, char *argv[]){
2      ...
3      for (i = 0; i < ITERATIONS; i++) {
4          clReleaseEvent(event_convo_row);
5          clEnqueueNDRangeKernel(main_command_queue, kernel_convo_row, 2, NULL, global_size,
               local_size, 1, &event_write, &event_convo_row);
6          clFlush(main_command_queue);
7
8          HostTask( getFrame, mem_img_orig );
9
10         clReleaseEvent(event_write);
11         clEnqueueWriteBuffer(write_command_queue, mem_img_orig, CL_FALSE, 0, MATRIX_SIZE,
               (void *)p_pinned_img_orig, 1, &event_convo_row, &event_write);
12         clFlush(write_command_queue);
13
14         clSetKernelArg(kernel_convo_col, 0, sizeof(cl_int), &SIZE_0);
15         clSetKernelArg(kernel_convo_col, 1, sizeof(cl_int), &SIZE_1);
16         clSetKernelArg(kernel_convo_col, 2, sizeof(cl_mem), &mem_img_gx);
17         clSetKernelArg(kernel_convo_col, 3, sizeof(cl_mem), &mem_img_temp);
18         clSetKernelArg(kernel_convo_col, 4, sizeof(cl_mem), &mem_kernel_gxc);
19
20         clReleaseEvent(event_convo_col1);
21         clEnqueueNDRangeKernel(main_command_queue, kernel_convo_col, 2, NULL, global_size,
               local_size, 0, NULL, &event_convo_col1);
22         clFlush(main_command_queue);
23
24         clSetKernelArg(kernel_convo_col, 0, sizeof(cl_int), &SIZE_0);
25         clSetKernelArg(kernel_convo_col, 1, sizeof(cl_int), &SIZE_1);
26         clSetKernelArg(kernel_convo_col, 2, sizeof(cl_mem), &mem_img_gy);
27         clSetKernelArg(kernel_convo_col, 3, sizeof(cl_mem), &mem_img_temp2);
28         clSetKernelArg(kernel_convo_col, 4, sizeof(cl_mem), &mem_kernel_gxr);
29
30         clReleaseEvent(event_convo_col2);
31         clEnqueueNDRangeKernel(main_command_queue, kernel_convo_col, 2, NULL, global_size,
               local_size, 1, &event_read, &event_convo_col2);
32         clFlush(main_command_queue);
33
34         clReleaseEvent(event_saxpy);
35         clEnqueueNDRangeKernel(main_command_queue, kernel_saxpy, 2, NULL, global_size,
               local_size, 0, NULL, &event_saxpy);
36         clFlush(main_command_queue);
37
38         clReleaseEvent(event_read);
39         clEnqueueReadBuffer(read_command_queue, mem_img_gy, CL_FALSE, 0, MATRIX_SIZE,
               (void *)p_pinned_img_gy, 1, &event_saxpy, &event_read);
40         clFlush(read_command_queue);
41
42         HostTask( putFrame, mem_img_gy );
43     }
44     ...
45 } //main
```

Figura 10: Ejemplo de código del filtro de Sobel en modo asincrono usando la interfaz de programación de OpenCL. Sólo se muestra el bucle principal de ejecución.

impacto de dicha sobrecarga en el rendimiento de la aplicación, hasta poder considerarse prácticamente despreciable.

Por último, se ha hecho patente la importante disminución que se puede conseguir en el esfuerzo de desarrollo necesario usando *Controllers* para programar aplicaciones paralelas, ya que la gestión transparente de las dependencias entre tareas conlleva una reducción en el número de líneas de código y en la complejidad de las mismas.

7

CONCLUSIONES

Este capítulo contiene:

- Revisión de los objetivos propuestos.
- Trabajo futuro.
- Valoración personal.

7.1 REVISIÓN DEL GRADO DE COMPLETITUD DE LOS OBJETIVOS PROPUESTOS

Este trabajo amplía las funcionalidades de la librería *Controllers*, uno de los principales proyectos de investigación del Grupo de Investigación Trasgo del Departamento de Informática de la Universidad de Valladolid. En concreto, en él se presenta un nuevo modelo de ejecución asíncrona para *Controllers*, capaz de analizar las dependencias entre cualquier combinación de tareas de transferencia de datos y/o de computación y, en función de dicho análisis, maximizar el solapamiento de las mismas con la menor sobrecarga posible.

Su implementación en OpenCL hace posible ejecutar aplicaciones programadas mediante la librería *Controllers* en sistemas heterogéneos sin modificaciones en el código. Además, el uso de esta librería para desarrollar aplicaciones paralelas supone una importante mejora del esfuerzo de desarrollo para el usuario, atendiendo a las métricas de número de líneas y complejidad ciclomática expuestas en el capitulo anterior.

Por tanto, se consideran satisfechos los siguientes objetivos específicos definidos en el capítulo introductorio de la presente memoria:

- Se ha analizado y entendido un modelo previo de comunicación asíncrona para dispositivos de tipo GPU, implementado de forma nativa para co-procesadores de un fabricante concreto (arquitectura CUDA de NVIDIA).
- Se ha propuesto un modelo alternativo para computación heterogénea.

- Se ha implementado el nuevo modelo propuesto utilizando OpenCL.
- Se ha validado experimentalmente el nuevo modelo propuesto.
- Se ha realizado un estudio experimental de su impacto en aplicaciones reales.

Finalmente, el resultado de este proyecto de investigación ha sido incluido en una artículo del Grupo Trasgo que ha sido recientemente enviado para su revisión por pares y eventual presentación en la edición de 2019 del congreso internacional *Principles and Practice of Parallel Programming* (PPoPP 2019). Este congreso se sitúa entre los más importantes a nivel mundial en el ámbito de la computación de altas prestaciones y la programación paralela, tal y como confirma su elevada calificación en la lista GGS (Clase 1 y Rating A+, equiparable a revistas Q1-Q2 JCR).

7.2 TRABAJO FUTURO

Durante la realización del presente trabajo fin de grado se han ido vislumbrando como especialmente prometedoras las siguientes dos líneas de trabajo futuro:

- Refinamiento del análisis y evaluación que se hace de las dependencias entre las tareas gestionadas por la cola, de modo que sea posible desacoplar el orden de evaluación y el de ejecución y así descubrir escenarios que permitan un mayor solapamiento.

- Aunque existe un gran número de dispositivos *stand-alone* y co-procesadores que pueden ser programados usando OpenCL, y por tanto ahora podrían ser programados también mediante Controllers, sería interesante incorporar nuevas implementaciones del modelo que permitan su ejecución en plataformas que necesitan de mecanismos propios de programación para poder explotar sus recursos al máximo, como es el caso de las últimas versiones de los co-procesadores Xeon Phi de Intel.

7.3 VALORACIÓN PERSONAL

En lo personal este trabajo ha supuesto un salto técnico importante. El estudiar un modelo teórico y buscar la forma de mejorarlo, desarrollando con fines de investigación, sin unos requisitos externos y rígidos que cumplir supone un cambio con todo lo que había hecho hasta ahora.

También he adquirido una serie de conocimientos y competencias con tecnologías complejas para la extracción de paralelismo como son CUDA y OpenCL, teniendo que indagar profundamente en sus interfaces de programación y en los detalles de implementación que permiten exprimir el rendimiento obtenido con el uso de las mismas.

Así mismo, gracias a su realización he podido conocer el mundo de la investigación académica, integrándome durante unos meses en la vida diaria del Grupo Trasgo y participando activamente en la elaboración de un artículo para su envío a un congreso de primer nivel. Esto ha supuesto un giro total en mi planteamiento futuro, habiendo tomado la decisión de cursar un máster en la rama de HPC como primer paso hacia una futura carrera investigadora centrada en este trabajo.

BIBLIOGRAFÍA

[1] OpenCL - The open standard for parallel programming of heterogeneous systems, July 2013.

[2] AMD. OpenCL: The GPU Open Compute Language.

[3] Juan Felipe Silva Garces. Computación heterogénea y su gran auge en los últimas décadas.

[4] Arturo Gonzalez-Escribano and Diego R Llanos. An Extensible System for Multilevel Automatic Data Partition and Mapping. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 25(5):10, 2014.

[5] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[6] M. Mercier, D. Glesser, Y. Georgiou, and O. Richard. Big data and HPC collocation: Using HPC idle resources for Big Data analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 347–352, December 2017.

[7] Hans Werner Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. *The TOP500: History, Trends, and Future Directions in High Performance Computing*. Chapman & Hall/CRC, 1st edition, 2014.

[8] Ana Moreton, Hector Ortega, and Arturo Gonzalez. Controllers: An abstraction to ease the use of hardware accelerators. *The International Journal of High Performance Computing Applications*, page 16.

[9] Ana Moreton-Fernandez, Eduardo Rodriguez-Gutiez, and Diego R Llanos. Supporting the Xeon Phi coprocessor in a Heterogeneous Programming Model. page 13.

[10] NVIDIA Corporation. CUDA C Programming Guide.

[11] Ismael J. Taboada, Yuri Torres, and Arturo Gonzalez-Escribano. Solapamiento transparente de tareas de comunicación y computación para mejor rendimiento de aplicaciones de GPU. page 9.

[12] Umesh Tiwari and Santosh Kumar. Cyclomatic Complexity Metric for Component Based Software. *SIGSOFT Softw. Eng. Notes*, 39(1):1–6, February 2014.

Apéndices

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.