TRABAJO FIN DE MÁSTER
MÁSTER INTERUNIVERSITARIO EN
COMPUTACIÓN DE ALTAS PRESTACIONES

Modelo de ejecución y sincronización en múltiples dispositivos heterogéneos

Estudiante: Sergio Alonso Pascual
Director/a/es/as: Arturo González Escribano
Diego Andrade Canosa
A Coruña, 31 de enero de 2023.

Agradecimientos

A mi familia y a mis tutores por ayudarme hasta el final a pesar de los contratiempos y adversidades. Gracias.

Resumen

Actualmente, la programación paralela basa sus soluciones sobre todos los tipos de hardware existentes. Esto da lugar a plataformas heterogéneas que hacen uso de coprocesadores. Uno de los desafíos que tiene programar eficientemente una aplicación para estos sistemas es la gestión de memoria. Los coprocesadores tienen su propio espacio de memoria, separado de la memoria de la máquina donde están instalados.

En este contexto, surgen diversas propuestas para facilitar al programador el uso de estos recursos. Una de estas propuestas es el modelo Controller del grupo de investigación Trasgo, que permite el solapamiento de operaciones de comunicación y computación en aceleradores hardware, así como gestión automática de las transferencias de memoria entre el host y el acelerador hardware. En la actualidad, el modelo Controller soporta el uso de dispositivos de tipo CPU (usando OpenMP), GPUs (usando CUDA y OpenCL) y FPGAs (usando OpenCL). Pero no el uso de varios dispositivos desde el mismo proceso.

Este trabajo propone una extensión del modelo de programación Controller para permitir el uso de varios dispositivos en el mismo proceso. También se realiza un estudio experimental para medir el rendimiento del nuevo modelo.

Abstract

Currently, parallel programming bases its solutions on all types of existing hardware. This results in heterogeneous platforms that make use of coprocessors. These systems present a series of challenges when developing applications using these devices efficiently, one of them is memory management. Coprocessors have their own memory space, separate from the memory of the machine where they are installed.

In this context, various proposals arise to make it easier for the programmer to use these resources. One of these proposals is the Controller model developed by Trasgo research group which allows the overlapping of communication and computing operations in hardware accelerators as well as automatic management of memory transfers between the host and the accelerator. Currently, the Controller model supports the use of CPU devices (through OpenMP), GPU devices (through CUDA and OpenCL) and FPGA devices (through OpenCL). But not the use of multiple devices in the same process.

This work proposes an extension of the Controller programming model to allow the use of multiple devices in the same process. An experimental study is also done to measure the performance of the new backend.

Palabras chave:

- Computación de alto rendimiento
- Sistemas heterogéneos
- Programación paralela
- Multicore
- Aceleradores hardware
- Multi-GPU
- FPGA

Keywords:

- High performance computing
- Heterogeneous systems
- Parallel programming
- Multicore
- Hardware accelerators
- Multi-GPU
- FPGA

Índice general

Índice de figuras

Índice de cuadros

Listings

Capítulo 1

Introducción

1.1 Contexto

La computación heterogénea se refiere al uso de unidades de cómputo de naturaleza diferente. La computación heterogénea se utiliza para aprovechar, en la medida de los posibles, todos los recursos hardware del sistema en la ejecución de una aplicación. La computación heterogénea se presenta como una solución para conseguir supercomputadores cada vez más rápidos capaces de resolver problemas más grandes y complejos en ámbitos como la ciencia y la ingeniería.

En la lista TOP500 [2] se encuentran los 500 supercomputadores más potentes del mundo en la actualidad. La mayoría de estos supercomputadores incluyen coprocesadores de alto rendimiento, también conocidos como aceleradores hardware. Estos aceleradores son de muchos tipos, tales como Unidades de Procesamiento Gráfico (GPUs) [3] o FPGAS (*Field Programable Gate Arrays*) [4]. La programación para este tipo de dispositivos es una tarea compleja que requiere de amplios conocimientos por parte del programador.

Muchos modelos de programación para desarrollar aplicaciones que usen los recursos hardware de un coprocesador utilizan el concepto de kernel: una unidad de código que debe ser compilada para un dispositivo concreto, y puede lanzarse desde el programa principal (programa host). Internamente, los coprocesadores hacen uso de sus recursos hardware para explotar, en la medida de lo posible, el posible paralelismo inherente al código del kernel.

Los datos sobre los que trabajan los kernels deben transferirse entre la memoria del host (la máquina que aloja al acelerador) y el coprocesador, ya que sus espacios de memoria son distintos. La transferencia de datos entre el espacio de memoria del host y del coprocesador es una de las tareas que más tiempo requiere. Una forma de minimizar los sobrecostes asociados con múltiples transferencias de este tipo es agrupar las transferencias cuando sea posible para

mover todos los datos necesarios en una sola transferencia.

En muchas aplicaciones encontramos transferencias de memoria y llamadas a kernels de forma intercalada. Realizar estas operaciones de forma secuencial introduce, en ocasiones, retrasos significativos en la ejecución de las aplicaciones que podrían ser reducidos, haciendo uso de las transferencias asíncronas que muchos modelos de programación soportan. Las operaciones asíncronas permiten que operaciones de cómputo y de comunicaciones que lanza el host se ejecuten de forma simultánea. El mayor inconveniente de estas llamadas es que obligan al programador a usar complejas técnicas de bajo nivel, en muchas ocasiones específicas del modelo de programación utilizado, limitando la portabilidad de los programas creados a otro tipo de aceleradores. Además de complicar aún más el trabajo del programador al requerir un análisis cuidadoso de la aplicación concreta para determinar cuando se puede obtener una mejora de rendimiento con estas técnicas y para evitar errores en el manejo de las operaciones de sincronización, que pueden ser difíciles de detectar y depurar.

Usar los coprocesadores para implementar aplicaciones asíncronas de forma eficiente es una tarea costosa, ya que requiere un alto conocimiento de los detalles de los diferentes recursos hardware del dispositivo y de mecanismos de programación específicos. Para ello, se han propuesto múltiples herramientas que introducen abstracciones donde el programador no necesita conocer dichos detalles de bajo nivel. Pudiendo desarrollar aplicaciones explotando el paralelismo, asincronía y posible solapamiento inherente de estas sin necesidad de gestionar o seleccionar parámetros específicos de la arquitectura.

1.2 Motivación

Controller [1, 5] es un modelo de programación paralela desarrollado por el grupo Trasgo [6] para reducir el esfuerzo de desarrollo de aplicaciones paralelas utilizando aceleradores. Este modelo proporciona una forma sencilla de enviar kernels a distintos tipos de aceleradores, automatizando operaciones tales como la selección de los atributos de lanzamiento del kernel, las transferencias de datos entre host y acelerador o solapando automáticamente aquellas tareas independientes entre sí.

Actualmente, Controller soporta varias plataformas y tipos de aceleradores. No obstante, una limitación actual del modelo Controller, es que dentro de un proceso solo puede existir un objeto controlador al mismo tiempo y, por lo tanto, solo es posible gestionar un dispositivo. La gestión de varios dispositivos en la misma máquina se tiene que realizar lanzando varios procesos coordinados en la misma. El objetivo de este trabajo de fin de máster consiste en tratar de eliminar esta restricción del modelo Controller para permitir la gestión de múltiples dispositivos heterogéneos al mismo tiempo en el mismo código mediante múltiples objetos

controladores.

1.3 Objetivos

Este trabajo propone una mejora del modelo de programación Controller [1, 5] para permitir su uso en aplicaciones o casos que pueden beneficiarse del uso de múltiples aceleradores heterogéneos. Esta nueva versión debe mantener la funcionalidad ya existente y extenderla al uso de varios dispositivos simultáneamente.

Los objetivos específicos de este trabajo son los siguientes:

1. Estudiar y evaluar el estado actual del modelo Controller.
2. Estudiar y evaluar soluciones similares en el estado del arte.
3. Diseñar un sistema que permita coordinar objetos controladores de diferentes arquitecturas.
4. Implementación de dicho sistema en el prototipo del modelo Controller.
5. Diseño e implementación de una aplicación que permita mostrar las nuevas funcionalidades del modelo Controller.
6. Pruebas experimentales.
7. Obtención y análisis de resultados.
8. Obtención de conclusiones.

1.4 Metodología

El proyecto se ha realizado usando una metodología incremental debido a su flexibilidad y eficiencia. Esta metodología se basa en el enfoque científico y consiste en estudiar, desarrollar y evaluar pequeños incrementos de un proyecto en lugar de tratar de entregar todo el proyecto de una vez.

El enfoque científico es esencial en el desarrollo de un proyecto con metodología incremental, ya que implica una investigación sistemática y rigurosa para entender el problema y desarrollar soluciones efectivas paso a paso. Esto incluye la búsqueda y análisis de la bibliografía relevante para el proyecto y la realización de experimentos controlados para evaluar y perfeccionar las soluciones propuestas.

La búsqueda de bibliografía es un componente crucial, permitiendo obtener una comprensión profunda del problema y de las soluciones existentes. Esto nos permite identificar las fortalezas y debilidades de las soluciones existentes para desarrollar soluciones innovadoras

que aborden estas debilidades.

1.5 Estructura del documento

El resto del documento se divide en los siguientes capítulos: El capítulo 2 presenta trabajos relacionados, así como el estado de Controller al inicio del proyecto. El capítulo 3 describe la especificación del modelo propuesto para la versión multi-dispositivo de Controller. El capítulo 4 describe la implementación realizada del modelo sobre la librería Controller. El capítulo 5 expone los resultados experimentales obtenidos en los casos de estudio que permiten validar el modelo propuesto. Por último, el capítulo 6 resume las conclusiones obtenidas y objetivos cumplidos.

Capítulo 2

Estado del arte

En este capítulo se presentan otros modelos de programación paralela heterogénea, clasificándolos basándonos en sus funcionalidades. También se expone una descripción del estado actual del modelo Controller del Grupo Trasgo.

2.1 Modelos de programación paralela heterogénea

Durante los últimos años se han propuesto diversos modelos para la programación paralela heterogénea. Muchos de ellos introducen abstracciones con la intención de solapar transferencias de memoria entre host y dispositivos y computación. En esta sección veremos algunas de las diferentes propuestas, clasificándolas en base a como gestionan las transferencias de memoria y sincronizaciones.

2.1.1 Proyectos con transferencias de memoria y sincronizaciones explícitas

Aproximaciones de bajo nivel

Una opción bastante intuitiva para sacar la mayor ventaja posible de un sistema heterogéneo es diseñar y programar a mano una versión de la aplicación específicamente para un dispositivo concreto usando modelos de programación nativos u ofrecidos por el vendedor de hardware como CUDA [7], Level Zero [8], HIP [9] u OpenCL [10].

Esta forma, permite administrar de la forma más eficiente posible los recursos de hardware y configuración, pero el programador necesita un conocimiento avanzado de las arquitecturas con las que se desea trabajar, así como gestionar manualmente los mecanismos de sincronización. Además de esto, es poco probable que la aplicación resultante sea fácilmente portable a otras configuraciones de hardware.

Aproximaciones con transferencias de datos explícitas de bajo nivel

Otro enfoque posible es introducir abstracciones de programación de alto nivel, pero no para la gestión de transferencias de memoria, donde las llamadas de bajo nivel todavía son necesarias. Por ejemplo, OmpSs-2 [11] es un modelo de programación compuesto por un conjunto de directivas y rutinas de biblioteca que pueden ser utilizadas en conjunto con un lenguaje de programación de alto nivel para desarrollar aplicaciones paralelas. El flujo de control entre tareas se deriva implícitamente a partir de un análisis de dependencias de datos, como en OpenMP. No obstante, las transferencias de datos con Las GPU deben administrarse explícitamente con CUDA o una biblioteca similar de bajo nivel.

Modelos de alto nivel con transferencias y sincronizaciones de datos explícitas

Existen modelos de programación heterogéneos con mayores niveles de abstracción que proporcionan mecanismos portables para la comunicación y sincronización entre el host y los dispositivos. Algunos de ellos requieren la invocación explícita de estos mecanismos por parte del programador, incluida la gestión de operaciones asíncronas, colas, streams o conceptos similares.

Por ejemplo, Kokkos [12, 13] es un modelo de programación de C++ para escribir aplicaciones portables de alto rendimiento orientadas a todas las principales plataformas HPC. Está diseñado para ser usado en arquitecturas de nodos complejos con jerarquías de memoria de N niveles y elementos de cómputo de varios tipos. Sin embargo, el proceso de compilación requiere que se seleccione un tipo de dispositivo objetivo, el cual, no puede ser cambiado durante la ejecución.

Con respecto a las transferencias de datos, Kokkos está diseñado de forma que el sistema nunca determina dónde o cuándo se debe realizar una transferencia de datos para mantener la coherencia de la memoria entre diferentes dispositivos. El programador es el responsable de hacerlo, invocando explícitamente una función específica (deep-copy) para realizar la transferencia de datos. En Kokkos, solo el uso de mecanismos como la memoria unificada de CUDA o similares podría evitar las llamadas de copia profunda explícitas, pero es una solución no portable e introduce una penalización de rendimiento [14]. Además, cuando se utiliza la función de copia profunda para realizar una transferencia de datos a través de diferentes jerarquías de memoria, siempre implica una operación de tipo “fence” completa. Esto significa que el sistema de ejecución introduce una sincronización de las colas de comandos antes y después de la transferencia de datos para mantener la consistencia de la memoria. [15]. Por lo tanto, las operaciones de transferencia de datos en Kokkos siempre son síncronas y el modelo de programación no admite transferencias de datos superpuestas con cálculos en el mismo dispositivo.

HPX [16, 17] es un sistema de tiempo de ejecución paralelo que amplía el estándar C++11 para facilitar las operaciones distribuidas, permitir un paralelismo detallado basado en restricciones y admitir la gestión de recursos adaptables en tiempo de ejecución. El programador debe realizar explícitamente la gestión de datos en dispositivos GPU utilizando un modelo de programación de nivel inferior, como CUDA. Incluye mecanismos de sincronización explícitos, incluida una abstracción de cola para transferencias de datos.

Otros ejemplos en esta categoría incluyen dCUDA [18], Groute [19], BlasX [20], GCharm [21] o Executors [22].

2.1.2 Proyectos con transferencias de memoria y sincronizaciones implícitas

item

Planificación automática de patrones de bucles y otros enfoques basados en tareas

Algunas herramientas y bibliotecas proponen enfoques abstractos que están orientados hacia la ejecución automática y paralela de bucles en dispositivos heterogéneos. Por ejemplo, Raja [23, 24] es una capa de abstracción de C++, desarrollada en el Lawrence Livermore National Laboratory (LLNL), que tiene como objetivo permitir la portabilidad del rendimiento. Trabaja sobre el paralelismo a nivel de bucle para las aplicaciones de C++ y se basa únicamente en las funciones estándar del lenguaje C++11 para su interfaz externa. Raja tiene extensiones internas que usan OpenMP, CUDA y AMD HIP para administrar dispositivos heterogéneos. Otros ejemplos incluyen LogFitc [25] o Maat [26].

También hay estrategias para problemas específicos, como el enfoque híbrido de CPU/GPU descrito en [27] para cálculos iterativos de stencil, que introducen comunicaciones asíncronas y equilibrio de carga entre dispositivos. También existen enfoques más genéricos orientados a tareas con soporte para aceleradores de GPU que derivan automáticamente dependencias para construir un grafo de tareas en tiempo de ejecución, utilizando técnicas sofisticadas de análisis de grafos para planificar tanto las tareas como las transferencias de datos necesarias [28]. Los modelos de esta categoría no admiten la coordinación de grafos de tareas de cualquier tipo, con dependencias de datos genéricas, como los generados en diferentes bucles anidados.

Enfoques de programación heterogéneos, genéricos y de alto nivel

Varios modelos utilizan abstracciones de programación más genéricas, tratando de lograr la portabilidad tanto del código como del rendimiento en la programación heterogénea. Algunos de ellos se implementan aprovechando en su interfaz características modernas de C++.

Por ejemplo, SYCL [29] es un estándar para la programación multiplataforma. Los kernels están organizados por un grafo de tareas que está implícitamente construido en tiempo de ejecución. El flujo de control y las comunicaciones de datos también pueden ser implícitos. El ecosistema SYCL actualmente contiene cuatro implementaciones principales de SYCL: ComputeCpp [30] de Codeplay, oneAPI [31] de Intel, triSYCL [32] y hipSYCL [33]. En general, las implementaciones actuales se basan en backends diferentes y no compatibles para diferentes tipos de dispositivos. TriSYCL solo admite CPU y FPGA de Xilinx. HipSYCL admite CPU y GPU de diferentes proveedores, pero no FPGA. ComputeCpp admite CPU y GPU NVIDIA. Con respecto a oneAPI, solo admite la combinación de núcleos de CPU Intel con GPU Intel e Intel FPGA. Existe un proyecto para soportar dispositivos NVIDIA, pero usando un backend de CUDA alternativo que aprovecha la infraestructura LLVM [12]. Así, todas estas implementaciones tienen limitaciones para operar con ciertas combinaciones de dispositivos.

Otro modelo en esta categoría es dOCAL [34]. Presenta una API de abstracción de alto nivel en C++ para simplificar la implementación de programas OpenCL/CUDA distribuidos. Administra y minimiza automáticamente las transferencias de datos. dOCAL es compatible con librerías de OpenCL y CUDA existentes; se puede conectar con sistemas de optimización automáticos y puede perfilar el comportamiento en tiempo de ejecución de los programas OpenCL y CUDA. También puede aprovechar el uso de la memoria unificada y la memoria "pinned" que puede acelerar, ocultar o incluso evitar las transferencias de datos entre las memorias de los dispositivos y la memoria principal.

Todos estos modelos abogan por el uso de un enfoque de código único para múltiples dispositivos. Esto incluye la encapsulación de tareas de CPU en kernels, para aprovechar las capacidades de ejecución en paralelo. Por lo tanto, estos núcleos no deben incluir código restringido a las CPU, como la gestión de E/S, o llamadas a bibliotecas de terceros específicas para un dispositivo en particular, como las que se usan en las aplicaciones de streaming de vídeo. Estos son ejemplos de escenarios en los que este tipo de operaciones son difíciles de sincronizar con otros núcleos y con las transferencias de datos utilizando un análisis de dependencia de datos implícito.

2.2 El modelo Controller

El modelo de programación paralela heterogénea Controller [1, 5, 35] se utiliza como punto de partida de este trabajo.

Controller es un modelo de programación paralela heterogénea que permite portabilidad en dispositivos de tipo CPU (usando OpenMP), GPUs (usando CUDA y OpenCL) y FPGAs (usando OpenCL). Está implementado como una librería escrita en C99. Por lo tanto, es com-

patible con cualquier compilador C99/C++ y es fácilmente interoperable con otras librerías y modelos de programación paralela.

2.2.1 Arquitectura de Controller

Controller propone un objeto abstracto para coordinar de manera transparente las actividades de ejecución y gestión de memoria en un acelerador o un conjunto de núcleos de CPU. En la figura 2.1 se pueden ver los elementos del modelo Controller. Un objeto controlador está asociado a una instancia particular de un dispositivo que se especifica en la llamada utilizada para su creación y gestiona de forma transparente la coordinación y comunicación del host con ese dispositivo.

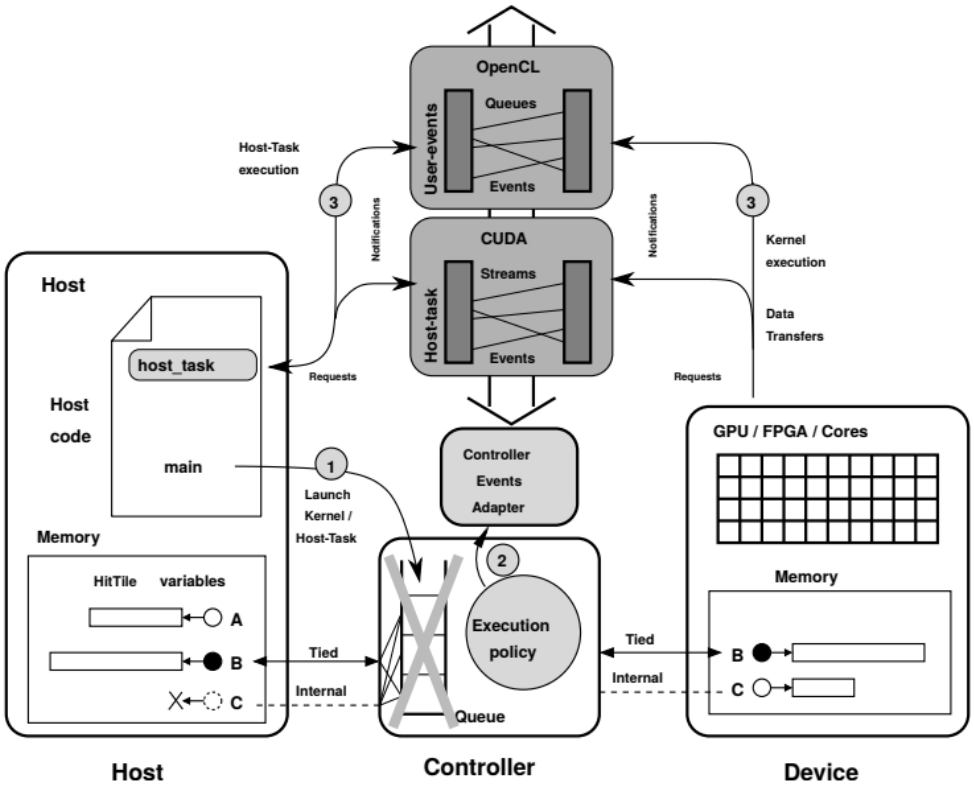

Figura 2.1: Diagrama de la arquitectura del modelo Controller. [1]

El soporte para cada tipo de dispositivo se da mediante un backend para cada uno de los tipos correspondientes, actualmente CPU, CUDA, OpenCL-GPU y FPGA. Los programas se pueden compilar con soporte para múltiples backends que se integran en el mismo sistema de ejecución, permitiendo así, elegir diferentes dispositivos objetivo en tiempo de ejecución.

Controller permite múltiples implementaciones para cada kernel, con versiones especializadas para cada tipo de dispositivo si es necesario. Las implementaciones del kernel se com-

pilan con el compilador nativo o las herramientas específicas ofrecidas por el proveedor del dispositivo objetivo.

El modelo ofrece un tipo de datos para encapsular variables y estructuras de datos (ej. arrays multidimensionales). Llamaremos a las variables de este tipo "tiles". Las tiles se crean mediante la función `Ctrl_Domain` que recibe entre otros parámetros un controlador al que la tile queda asociada. Es posible especificar que la tile tenga memoria tanto en el host como en el dispositivo del controlador (*tied*) o solo en el dispositivo del controlador (*internal*).

En el backend de CPU existe una opción para usar tiles con una única imagen de memoria que se utiliza tanto para kernels como para tareas de host. Llamamos a este modo u opción "0 copy", ya que permitirá utilizar los datos de la tile tanto en kernels de CPU como en el código principal de coordinación sin necesidad de realizar transferencias de datos.

En Controller las interacciones entre el host y el dispositivo se interpretan como una serie de operaciones controladas por el código de coordinación ejecutado en el host. Una operación puede ser de uno de los siguientes tipos:

- `Alloc(x)`. Reserva memoria para la tile *x* creada previamente. Es posible controlar si la tile será *tied* o *internal* así como si la memoria del host será *pinned* mediante un parámetro opcional. Esta operación siempre es síncrona.
- `Free(x)`. Libera las imágenes de memoria de la tile *x*. Esta operación siempre es síncrona.
- `HTD(x)` Transfiere los datos de la tile *x* del espacio de memoria del host al espacio de memoria del dispositivo.
- `DTH(x)` Transfiere los datos de la tile *x* del espacio de memoria del dispositivo al espacio de memoria del host.
- `Kernel(f, th, args)`. Una petición para ejecutar un kernel en el dispositivo. Recibe un nombre de función *f*, un bloque de hilos *th* con los que ejecutar el kernel y los argumentos que recibe el kernel *f*.
- `Host(h, args)`. Una petición para ejecutar la tarea de host *h*.
- `Wait(x)`. Una petición para bloquear la ejecución del código del host hasta que todas las operaciones que implican a la tile *x* hayan finalizado.
- `WaitAll()` Una operación de espera global, espera hasta que todas las operaciones que involucren alguna tile del controlador hayan finalizado.

El módulo de política de ejecución se ocupa de procesar las tareas que llegan al controlador. Para esto, se transfieren las operaciones al backend correspondiente al controlador usando una API genérica. Cada backend ejecuta las operaciones usando los modelos de programación de

bajo nivel del dispositivo correspondiente.

El modelo Controller soporta 2 políticas de ejecución, síncrona y asíncrona. En la política síncrona todas las operaciones suceden de forma secuencial, sin solaparse. Mientras que en la política asíncrona es posible que se solapen operaciones si las dependencias lo permiten.

2.2.2 La librería de tiling: Hitmap

Hitmap [36] es la librería utilizada en el modelo de Controller para proporcionar una interfaz común abstracta para la gestión de datos tanto en el código host como en los kernels ejecutados en diferentes tipos de dispositivos.

Las estructuras de datos en el host se administran con objetos HitTile, que son "punteros gordos" que almacenan varios metadatos, tales como dimensiones, tamaños de datos y el puntero de datos. El modelo Controller extiende la estructura HitTile para almacenar nuevos metadatos relacionados con el uso de la tile en un dispositivo. Se incluye información para encontrar los datos en el dispositivo, eventos que permiten comprobar el estado de ejecución de operaciones previas e información sobre el estado de la memoria. La función para acceder a los datos de una tile de Hitmap, `hit()`, se utiliza en los códigos de host o kernel para acceder a los elementos de una tile. De esta forma Hitmap proporciona una vista portable, con colocación de datos en la memoria siguiendo un orden *row-major* en cualquier dispositivo.

2.2.3 Dependencias entre operaciones asíncronas

En la política asíncrona, las tareas del host y los kernels se pueden solapar si las dependencias lo permiten (preservando sus órdenes parciales), y las transferencias de datos se pueden solapar con la ejecución de los kernels y las tareas del host. Las reglas internas que deciden cuándo una solicitud se puede solapar con otras se diseñan estudiando las dependencias entre los diferentes tipos de operaciones y teniendo en cuenta el papel de entrada/salida de sus parámetros. En la tabla 2.1 se pueden ver las reglas de dependencias entre las diferentes operaciones del modelo Controller.

2.2.4 Comunicaciones implícitas

Usando los nuevos metadatos introducidos en las estructuras HitTile originales para registrar el estado de los eventos y la información de consistencia de la memoria, podemos verificar y modificar directamente el estado de un HitTile en tiempo de ejecución.

Es posible detectar la necesidad de emitir operaciones de comunicación implícitas para mantener la consistencia de la memoria entre las imágenes del host y del dispositivo cuando se evalúan las operaciones del controlador. Mientras se evalúa una tarea del kernel o del

Op \ Waits to	K(In x)	K(Out x)	H(In x)	H(Out x)	HTD(x)	DTH(x)
K(In x)		x			x	
K(Out x)	x	x			x	x
H(In x)				x		x
H(Out x)			x	x	x	x
HTD(x)	x	x		x		x
DTH(x)		x	x	x	x	
Free(x)	x	x	x	x	x	x
Wait(x)	x	x	x	x	x	x

Cuadro 2.1: Dependencias entre operaciones de Controller aplicadas sobre una tile *x*. En las operaciones de kernel (*K*) y tareas de host (*H*) distinguimos entre los casos en los que *x* es un parámetro de entrada (In) o salida (Out). Una marca en la celda significa que la operación de esa fila deberá esperar a la operación de la columna correspondiente.

host en busca de dependencias, los metadatos de las tiles se actualizan. Estas actualizaciones expresan el estado futuro de los parámetros relacionados con la consistencia de la memoria y los eventos de sincronización tras la ejecución de la operación, después de que se cumplan todas las condiciones de espera. Por lo tanto, la próxima vez que se use la tile, durante la evaluación de una operación, se puede determinar si es necesaria una transferencia de datos para garantizar la consistencia de la memoria. En la tabla 2.2 se pueden ver las diferentes acciones basándose en los estados de la tile.

operación	Sh(x)	Sd(x)	acciones
K(In x)	0	0	Warning
	0	1	-
	1	0	sd(x) = 1; HTD(x)
	1	1	-
K(Out x)	0	0	sd(x) = 1
	0	1	-
	1	0	sh(x) = 0; sd(x) = 1; HTD(x)
	1	1	sh(x) = 0
H(In x)	0	0	Warning
	0	1	sh(x) = 1; DTH(x)
	1	0	-
	1	1	-
H(Out x)	0	0	sh(x) = 1
	0	1	sh(x) = 1, sd(x) = 0; DTH(x)
	1	0	-
	1	1	sd(x) = 0

Cuadro 2.2: Reglas para las comunicaciones implícitas del modelo Controller. Sd y Sh indican el estado de la tile *x*, 0 inválido, 1 válido.

2.2.5 Ejemplo de uso de Controller

Declaración de kernels

En el modelo de Controller, un kernel se declara utilizando dos macros.

El primero es CTRL_KERNEL_PROTO, que declara un prototipo para todas las implementaciones de un kernel dado. Ver líneas 8-13 en el listing 2.1. El primer parámetro del macro es el nombre del kernel, seguido por el número de implementaciones y sus tipos y subtipos, finalmente se indican el número de argumentos que recibe el kernel, así como sus roles (*INVAL* para argumentos por valor, *IN* para argumentos de lectura, *OUT* para escritura o *IO* para lectura y escritura), tipos y nombres. Controller lo utiliza para localizar, en tiempo de ejecución, la mejor implementación disponible de un kernel para el dispositivo asociado.

Cada implementación de kernel se declara utilizando el macro CTRL_KERNEL. Ver líneas 1-5 en la figura 2.1. Los primeros parámetros son el nombre del kernel y el tipo y subtipo de la implementación. Después de los parámetros del kernel, el código se incluye en un bloque estructurado. Este ejemplo particular calcula una suma de matrices. La implementación es una implementación genérica paralela de grano fino, que el controlador puede ajustar y adaptar automáticamente a la granularidad de tareas adecuada de los diferentes tipos de dispositivos, como GPU, o conjuntos de núcleos de CPU.

Para determinar la posición que se va a calcular, se usan las variables `thread_id_x` y `thread_id_y` que genera Controller y toman diferentes valores para cada hilo lógico en una cuadrícula definida por el usuario. Esta solución es una alternativa portátil a threadIdx.x y threadIdx.y en CUDA. Los hilos fuera de la cuadrícula del usuario, agregados en dispositivos como GPU debido a su sistema de bloques, se omiten internamente antes de la ejecución del código de usuario. Controller permite trabajar con espacios de threads de 1 a 3 dimensiones.

Declaración de tareas de host

El modelo de Controller también permite la declaración de tareas de host que se ejecutarán asíncronamente en el host. Ver listing 2.2 La declaración de estas tareas de host es muy similar a la explicada en la sección anterior, salvo que se usan los macros CTRL_HOST_TASK y CTRL_HOST_TASK y el código que debe ser ejecutado es un bloque estructurado después del macro.

Operaciones de Controller

Todas las operaciones sobre un controlador se deben realizar dentro de un `ctrl_block` como se muestra en el listing 2.3. Entre otras tareas, este macro crea los hilos de OpenMP que se usan internamente durante el programa.

```
CTRL_KERNEL(Add, GENERIC, DEFAULT, KHitTile_float A, KHitTile_float B,
    KHitTile_float C,
{
  hit(C, thread_id_x, thread_id_y) =
      hit(A, thread_id_x, thread_id_y) +
      hit(B, thread_id_x, thread_id_y);
});

CTRL_KERNEL_PROTO( Add,
    1, GENERIC, DEFAULT,
    3,
    IN, HitTile_float, A,
    IN, HitTile_float, B,
    OUT, HitTile_float, C);
```

Listing 2.1: Ejemplo de prototipo e implementación de un kernel para suma de matrices usando la librería Controller.

```
CTRL_HOST_TASK(Norm_calc, HitTile_float matrix) {
    double res = 0;
    double sum = 0;
    for (int i = 0; i < hit_tileDimCard(matrix, 0); i++) {
        for (int j = 0; j < hit_tileDimCard(matrix, 1); j++) {
            sum += pow(hit(matrix, i, j), 2);
        }
    }
    res = sqrt(sum);
    printf("\n Result: %lf \n", res);
}

CTRL_HOST_TASK_PROTO(Norm_calc, 1, IN, HitTile_float, matrix);
```

Listing 2.2: Ejemplo de tarea de host para calculo de norma de una matriz con la librería Controller.

El macro recibe 2 parámetros, el primero es el número de controladores que se crearán en su interior (actualmente el límite es 1), y el segundo representa cuantos de esos controladores requerirán hilos internos. Actualmente, los únicos controladores que requieren hilos internos son los controladores de CPU para poder ejecutar los kernels de forma asíncrona.

2.3 Resumen

En la primera parte de este capítulo se han repasado diferentes soluciones para la programación heterogénea. Se ha observado que los modelos de programación estudiados no presentan mecanismos de implementación realmente portables que permitan la ejecución asíncrona y solapada de tareas y comunicaciones en varios dispositivos de tipos/fabricantes diferentes dentro del mismo programa. El modelo de Controller es una posible solución al permitir integrar en su sistema de ejecución diversos backends. Sin embargo, el diseño de sus mecanismos internos actualmente no permite construir más de un controlador en cada proceso. Resolver esta carencia sin impactar gravemente en el rendimiento implica estudiar posibles nuevas soluciones de sincronización, diseñar nuevos mecanismos portables de control y coordinación entre backends y tecnologías muy diferentes, así como resolver otros problemas de integración y portabilidad asociados. Estas nuevas soluciones son el objeto de este trabajo.

```
int main(int argc, char *argv[]){
    ...
    //Logical threads for the device
    Ctrl_Thread threads;
    Ctrl_ThreadInit(threads, rows, cols);
    __ctrl_block__(1, 0) {
        // 1. Create controller object
        PCtrl ctrl = Ctrl_Create(CTRL_TYPE_CUDA, policy, DEVICE);

        // 2. Alloc data structures
        HitShape shape = hitShapeSize(rows, cols);
        HitTile_float A = Ctrl_DomainAlloc(ctrl, float, shape);
        HitTile_float B = Ctrl_DomainAlloc(ctrl, float, shape);
        HitTile_float C = Ctrl_DomainAlloc(ctrl, float, shape);

        // 3. Initialize data structures
        Ctrl_HostTask(ctrl, Init_Tiles, A, B, C);

        // 4. Sync and start timer
        Ctrl_GlobalSync(ctrl);
        exec_clock = omp_get_wtime();

        // 5. Launch the kernel
        // Implicit: Ctrl_MoveTo(ctrl, A, B);
        Ctrl_Launch(ctrl, Add, threads, CTRL_THREAD_NULL, N_ITER, A, B, C);

        // 6. Sync and stop timer
        Ctrl_GlobalSync(ctrl);
        exec_clock = omp_get_wtime() - exec_clock;

        // 7. Calculate NORM
        // Implicit: Ctrl_MoveFrom(ctrl, C);
        Ctrl_HostTask(ctrl, Norm_calc, C);

        // 8. Free data structures
        Ctrl_Free(ctrl, A, B, C);

        // 9. Destroy the controller
        Ctrl_Destroy(ctrl);
    } //__ctrl_block__
    ...
}//main
```

Listing 2.3: Ejemplo del código del main de una suma de matrices usando la librería Controller. Las operaciones de movimientos de memoria que se realizan implícitamente se muestran con comentarios como se realizarían en el modo explicito.

Capítulo 3

Propuesta de solución

En este capítulo se describe la solución que proponemos para coordinar de forma eficiente varios dispositivos de naturaleza diferente en un sistema de programación y ejecución heterogénea. Es una solución que modifica y extiende el modelo actual de Controller para poder controlar varios dispositivos desde el mismo proceso. Se describen los problemas que encontramos durante el diseño, así como las diferentes posibles soluciones que se han explorado.

3.1 Idea general

La idea general para permitir el uso de múltiples dispositivos de forma simultánea es mediante el uso de varios controladores, con la posibilidad de tener tiles asociadas a varios controladores al mismo tiempo. Teniendo así estas tiles una imagen de memoria en el host y una en cada dispositivo. De esta forma, para realizar un movimiento de memoria de una tile de un dispositivo a otro, la tile debe estar asociada a ambos dispositivos y la transferencia se realizará a través del host.

Esto plantea varios problemas y opciones de diseño que consideraremos a continuación.

3.2 Sistema de eventos

Para permitir la comunicación asíncrona entre diferentes controladores deberemos encontrar una manera de hacer que los sistemas de eventos que usa cada backend sean compatibles los unos con los otros.

Por ejemplo, en el caso de que una tile esté asociada a 2 controladores (Ctrl A y Ctrl B) y sea necesario realizar una transferencia de un dispositivo al otro, esta transferencia debe realizarse a través del host. La secuencia sería algo así:

1. Ctrl A solicita una operación moveFrom al ctrl B.

2. Ctrl A realiza una espera hasta que la operación moveFrom haya terminado.
3. Ctrl A realiza una operación moveTo.

El problema surge en el segundo paso, ya que el Ctrl A debe esperar a eventos del Ctrl B y estos pueden ser de diferente tipo, implementados con diferentes tecnologías en backends diferentes. Esto es problemático, ya que cada backend usa los eventos y colas de su tecnología: el backend de CUDA usa eventos y streams de CUDA, los backends de OpenCL GPU y FPGA usan eventos y colas de comandos de OpenCL y el backend de CPU usa los eventos y las colas de tareas diseñadas e implementadas por nosotros.

Por lo general, estos eventos y colas no son compatibles los unos con los otros o en algunos casos incluso son incompatibles con eventos de su propia tecnología al usar dispositivos diferentes. Por ejemplo, en las colas de comandos de OpenCL solo se pueden usar eventos de OpenCL asociados al mismo contexto que la cola de comandos.

En CUDA, en un sistema de multi-dispositivo (varias GPUs de NVIDIA) las siguientes normas aplican [37]:

- `cudaEventRecord` fallará si el evento y el stream están asociados a diferentes dispositivos.
- `cudaEventSynchronize` y `cudaEventQuery` funcionarán incluso si el evento está asociado a un dispositivo diferente al actual.
- `cudaStreamWaitEvent` funcionará incluso si el stream y el evento están asociados a dispositivos diferentes. Esta función, por lo tanto, se puede usar para sincronizar múltiples dispositivos juntos.

En las colas de tareas de Controller (`Ctrl_TaskQueue`) podemos realizar esperas a eventos de CUDA, OpenCL y CPU usando "eventos genéricos". Estos eventos genéricos se proponen como un wrapper que nos permite utilizar una interfaz común con operaciones de espera a un evento, test de un evento y encolar una espera a un evento en una cola de tareas. También se permite la operación de señalar manualmente la finalización de un evento siempre y cuando el evento interno lo soporte. En el diagrama 3.1 se puede ver un diagrama de clases de los eventos genéricos.

3.2.1 Problemas en OpenCL

Idea 1: Hilo traductor de eventos

Una posibilidad para permitir a un controlador que usa OpenCL esperar a eventos de otro controlador es haciendo uso de eventos de usuario, creados mediante `clCreateUserEvent`. Estos eventos funcionan como los eventos de OpenCL normales, pero en vez de completar-

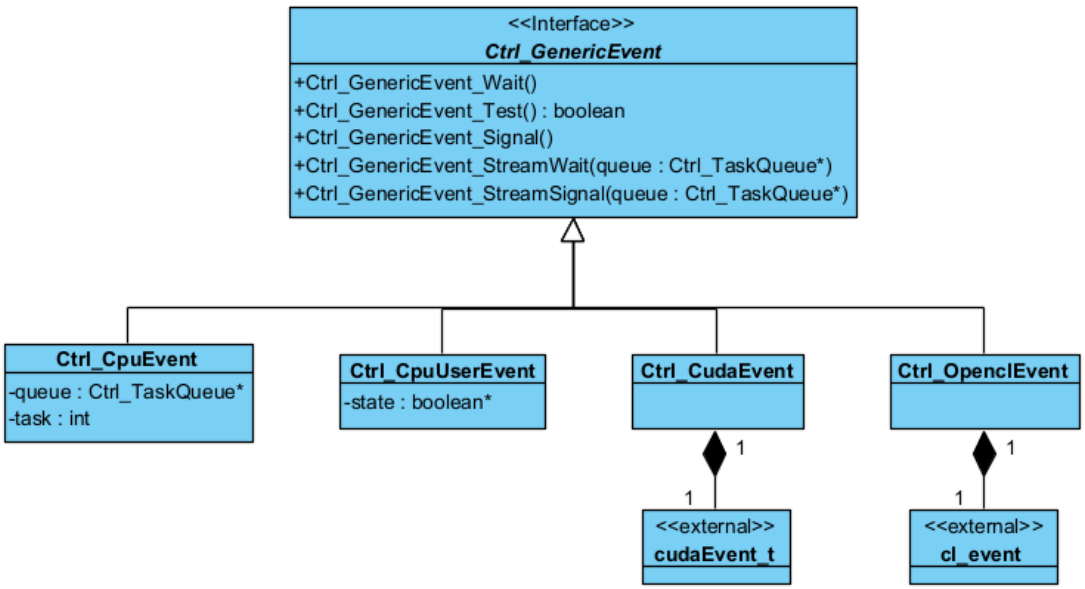

Figura 3.1: Diagrama de clases de los eventos genéricos en Controller.

se cuando se completa una operación en la cola de comandos correspondiente, se señala su finalización desde el host.

La idea sería usar un hilo adicional que se encargue de comprobar si ha finalizado el evento del controlador B que marca cuando la operación moveFrom se ha completado y en caso afirmativo señale un evento de usuario al que está esperando el controlador A.

Esta aproximación nos permitiría esperar desde una cola de OpenCL a eventos de cualquier tipo siempre y cuando dispongan de una función test. No obstante, el problema de esta aproximación es que no funciona para los streams de CUDA, ya que el concepto de eventos de usuario no existe en CUDA.

Idea 2: Usar un solo contexto de OpenCL para todos los dispositivos

Una forma de hacer que los eventos de OpenCL sean compatibles entre sí es usar un solo contexto que contenga todos los dispositivos que se vayan a usar.

El problema con esto es que OpenCL solo permite dispositivos pertenecientes a la misma plataforma en un contexto. Esto es un problema, ya que la idea es permitir el uso de múltiples dispositivos de diferentes vendedores.

Una solución podría ser usar una plataforma que agrupe todos los dispositivos, como por ejemplo POCL [38]. POCL soporta dispositivos de CPU, GPUs de Nvidia, dispositivos HSA (p. ej. GPUs de AMD), dispositivos TCE y fixed-function accelerators.

No obstante, esto requeriría depender del soporte de POCL, el cual está en estado experimental para algunos de estos dispositivos con los posibles problemas de rendimiento que ello puede conllevar. Además, esta solución no resuelve la interoperabilidad con CUDA.

3.2.2 Problemas en CUDA

Idea 1:Sincronización vía cudaLaunchHostFunc

En CUDA es posible encolar en un stream tareas que se ejecuten en un hilo de CPU mediante la función `cudaLaunchHostFunc`. Al usar esta funcionalidad, el resto de tareas en el stream no empezarán a ejecutarse hasta que la función del host haya terminado.

Es entonces posible, usar esto para señalar eventos de usuario de algún tipo cuando algo sucede en un stream CUDA o dejar bloqueado un stream de CUDA esperando a algún otro evento desde la función de host.

El problema con esta idea es que todas las funciones de host comparten un mismo hilo del driver, por lo tanto, utilizar esta idea para sincronizar múltiples streams al mismo tiempo es problemático.

Idea 2:CUDA Stream Memory Operations

En CUDA existe un concepto que podría ser usado para sincronizar streams de tal forma que se puedan desbloquear desde el host. Las Stream Memory Operations [39] son un grupo de operaciones de la API de bajo nivel del driver de CUDA mediante las cuales se puede dejar un stream bloqueado hasta que una dirección de memoria del dispositivo cambie de valor.

Estas operaciones funcionan de la siguiente manera:

- Las operaciones `cuStreamWaitValue` permiten dejar el stream bloqueado hasta que una dirección de memoria del dispositivo contiene un valor concreto.
- Las operaciones `cuStreamWriteValue` permiten encolar una escritura a una dirección de memoria del dispositivo.

Estas funciones se podrían usar en la forma de "eventos de usuario" en CUDA. Esto, unido a la idea discutida en la sección 3.2.1, podría dar una solución completa para todos los backends actuales.

El problema con esta solución es que este set de operaciones está desactivado por defecto en los drivers de NVIDIA hasta la versión 11.7.1 (agosto 2022). Para activarlo en linux era necesario pasar un parámetro concreto al módulo correspondiente mediante el comando `modprobe` y en otros sistemas operativos no era posible activar estas operaciones. Esta opción se descartó antes de que saliera esta versión y se comenzó la implementación de otra

solución, por lo tanto, la exploración de esta alternativa se dejará como trabajo futuro.

Idea 3:Implementación custom de eventos de usuario en CUDA

Una alternativa a usar las Stream Memory Operations como eventos de usuario de CUDA que consideramos es implementar un sistema de eventos de usuario basado en lo siguiente:

- Para bloquear el trabajo de un stream, usamos un kernel con 1 thread que compruebe una posición de memoria con operaciones atómicas hasta que obtenga el valor deseado.
- Para desbloquear el stream realizamos una transferencia de memoria de forma asíncrona desde otro stream.

El principal problema con esta opción es la posible perdida de rendimiento al tener que dedicar parte de la GPU que normalmente se usa para cómputo para gestionar estas sincronizaciones. Aparte de que esto requiere la ejecución simultánea de una cantidad considerable de kernels al mismo tiempo, lo cual podría acarrear problemas en algún punto.

3.2.3 Solución final: Sincronización con eventos en el host

En vista de los problemas mencionados anteriormente, parece que la mejor solución es mover todas las sincronizaciones y esperas a eventos a la CPU.

Cada cola del driver (streams de CUDA y colas de comandos de OpenCL), deberá tener entonces un equivalente en el host en la que se realizarán las esperas correspondientes. Una vez que se hayan realizado las esperas oportunas y una operación llegue al final de una cola del host, esta estará lista para ser ejecutada y, por lo tanto, se encolará en la cola del driver correspondiente.

Para la gestión de estas colas de host (esperas a los eventos introducidos e introducir las operaciones listas en las colas del driver), podríamos usar un hilo por cada cola. Pero esto sería muy costoso, ya que cada controlador hace uso de múltiples colas y podría llegar a usar muchos hilos. Por lo tanto, en vez de eso utilizaremos un solo hilo para la gestión de todas las colas de host. Llamaremos a este hilo "queue manager".

Para esto, en lugar de bloquear este hilo cuando encontramos una espera a un evento, utilizaremos una función de tipo "test" para conocer el estado del evento sin bloquear el hilo. Si el evento está completado podemos eliminarlo de la cola y pasar a la siguiente tarea, en caso contrario, simplemente pasaremos a comprobar la siguiente cola.

En la imagen 3.2 podemos ver un diagrama que muestra las nuevas colas e hilos usados en este nuevo sistema.

Como podemos ver, ahora las operaciones pasan por 2 colas, la del host y la del driver,

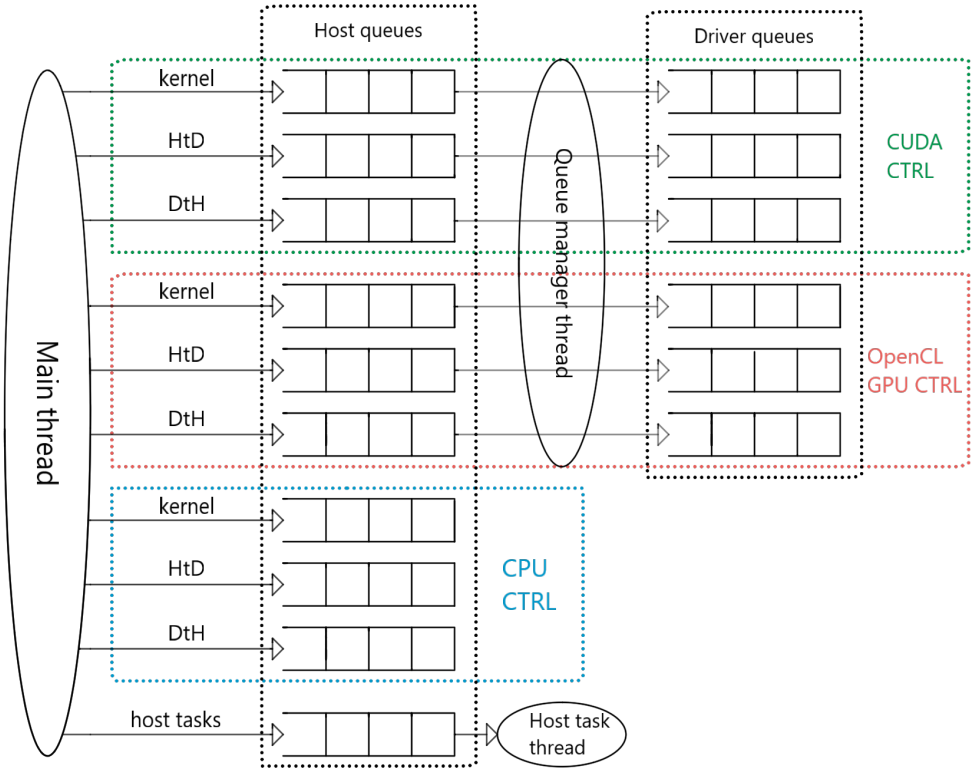

Figura 3.2: Diagrama mostrando los hilos y las colas con sus respectivas tareas en el modelo propuesto cuando se usan 3 controladores, uno de CPU, uno de CUDA y uno de OpenCL GPU.

esto hace que distingamos 2 periodos desde que encolamos una operación:

1. Cuando la tarea está en la cola del host. Desde que encolamos la tarea hasta que el "queue manager" la procesa y la encola en la cola del driver correspondiente.
2. Cuando la tarea está en una cola del driver. Desde que el "queue manager" procesa la tarea hasta que su ejecución finaliza.

Como ahora las tareas tienen estos 2 períodos distintos, cuando deseamos esperar a una tarea debemos usar 2 eventos. Uno correspondiente al 1er periodo que hace referencia a una cola de host y otro correspondiente al 2º periodo que hace referencia a una cola del driver.

Es necesario usar los 2 eventos y no solo el 2º, ya que este evento no se puede usar hasta que el hilo "queue manager" ha procesado la tarea y la ha introducido en la cola del driver.

Controladores de CPU y tareas de host

Todo esto no es necesario en el backend de CPU, ya que este funciona directamente con colas de host. Por lo tanto, cuando necesitemos esperar a un evento de otro backend, podemos

simplemente esperar a los 2 eventos genéricos asociados a esa tarea (host y driver). Cuando necesitemos esperar desde otro backend a una operación de CPU, podemos esperar a un solo evento genérico de tipo CPU que represente la operación.

Tampoco es necesario aplicar este modelo a las tareas de host, ya que, estas también funcionan directamente sobre colas de host. Por tanto, los eventos relacionados con las tareas de host serán eventos genéricos de tipo CPU (uno para lectura y otro para escritura). De hecho, estos eventos deben ser comunes para todos los controladores asociados a una tile.

Cuando necesitemos que una tarea de host espere a otra operación, simplemente esperará a los 2 eventos genéricos correspondientes (host y driver) o uno en el caso de que la tarea sea de CPU.

Consideraciones adicionales de esta propuesta

La decisión de mover todas las sincronizaciones al host es delicada, ya que es posible que cause una pérdida de rendimiento al perder las optimizaciones que realizan las tecnologías de bajo nivel en la gestión de sus colas y eventos. Por este motivo, antes de realizar una implementación completa decidimos realizar una prueba de rendimiento tentativa de este sistema. Durante esta prueba el sistema no parecía mostrar perdida de rendimiento, así que procedimos adelante con la implementación completa.

3.3 Problemas de portabilidad de los kernels

En Controller usar el mismo kernel para varios backends a la vez no es un problema, simplemente se compila para cada uno de los backends por separado y cada backend usa su versión.

No obstante, el uso de varios controladores del mismo backend, cada uno con su dispositivo, puede causar problemas en algunos casos.

En los backends de CPU y en CUDA esto no supone un problema, ya que varios dispositivos de estos backends pueden utilizar el mismo kernel compilado.

En cambio, sí que supone un problema para los backends de OpenCL GPU y FPGA, ya que en OpenCL para compilar un kernel debes primero crear un programa (*cl_program*) que va asociado al contexto, posteriormente este programa se compila para uno o varios dispositivos pertenecientes al contexto asociado.

Es decir, para usar un kernel de OpenCL para múltiples dispositivos, debemos hacer una de 2 cosas:

- Usar 1 solo contexto de OpenCL que contenga todos los dispositivos de ese backend.

- Compilar y almacenar una versión del kernel compilado para cada dispositivo del backend y que cada dispositivo use la suya.

Ya se han mencionado anteriormente 3.2.1 los problemas que tiene la primera opción, por lo tanto, usaremos la segunda.

3.4 Fichero de configuración

Con la solución que estamos proponiendo crear y destruir los controladores explícitamente en cualquier momento es complejo. Por ejemplo, un controlador puede tener asociada una imagen de memoria de un tile que está siendo sincronizada con otros controladores. Por otra parte, algunos recursos deben ser gestionados antes de la creación de los controladores. Principalmente, la creación de hilos necesarios para los controladores de CPU, tareas de host y la gestión de las colas mencionadas en la sección 3.2.3, así como algunas reservas de memoria (p. ej. los kernels de OpenCL mencionados en la anterior sección).

Esto se hace en el macro `__ctrl_block__` usando los argumentos (actualmente número de controladores en el bloque y número de controladores de CPU en el bloque) para saber qué hace falta crear. Con los nuevos cambios la información que hay que suministrar crece y se vuelve más compleja.

Por tanto, hemos decidido usar un fichero de configuración en el que se detalla la configuración de todos los controladores que se van a usar en el programa para suministrar la información necesaria al `ctrl_block` en tiempo de ejecución.

También hemos decidido mover la creación de todos los controladores a la implementación interna del macro `ctrl_block` para simplificar este proceso. De esta forma, el usuario no debe preocuparse de la creación o destrucción de los controladores, que queda especificada en el fichero de configuración escogido para la ejecución del programa.

3.5 Resumen

En este capítulo se han descrito las ideas fundamentales de nuestra propuesta para conseguir mecanismos de coordinación y sincronización eficientes que permitan al modelo Controller: (1) soportar la asociación de una misma estructura de datos (tile) a varios dispositivos; y (2) implementar sus políticas de ejecución síncrona o asíncrona para varios dispositivos de diferente naturaleza o fabricante. Esto permite en definitiva conseguir la portabilidad deseada, incluidos los mecanismos de comunicación implícita para mover datos entre dispositivos de forma coordinada cuando las dependencias entre tareas que usan los mismos tiles lo indiquen.

Capítulo 4

Implementación

En este capítulo se describe la implementación de la propuesta de solución realizada sobre el prototipo de Controller. Esta implementación se ha realizado de forma iterativa, comenzando por una versión básica y añadiendo funcionalidades de forma progresiva.

4.1 Múltiples controladores independientes

Esta primera versión consiste en permitir que existan múltiples controladores al mismo tiempo pero de forma independiente. Esto significa que:

- Las estructuras de tipo tile deberán pertenecer a un único controlador.
- Las operaciones deberán contener únicamente tiles pertenecientes al controlador usado para lanzar la operación.

Para lograr este objetivo, ha sido necesario hacer varios cambios en Controller que se detallan a continuación.

4.1.1 Lista de controladores singleton

Hasta ahora solo era posible tener un controlador al mismo tiempo. Este controlador se almacenaba en una variable global que funcionaba a modo de singleton para que los diferentes hilos tuvieran acceso a él.

Ahora que se permiten múltiples controladores simultáneamente, almacenamos una lista de los controladores, así como el tamaño de dicha lista en variables globales.

4.1.2 Hilos en la creación de los controladores de CPU

Como se ha mencionado en la sección 2.2.5, los controladores de CPU requieren la creación de hilos adicionales para ejecutar sus tareas. Ahora que podrán existir múltiples controladores,

la gestión de estos hilos plantea dos problemas, sincronizarlos para que esperen a la creación de su controlador y obtener su controlador de la lista global de controladores.

Sincronización de los hilos de CPU

Como los controladores se crean después del macro `__ctrl_block__` (en el cual se crean los hilos), es necesario hacer que los hilos esperen a la creación de su controlador.

Esta sincronización cuando solo había un controlador se hacía mediante una barrera, haciendo que todos los hilos esperaran a la creación del controlador.

Ahora que hay múltiples controladores, esta solución no es suficiente, cada hilo debe esperar a su controlador, no a los otros controladores. Esto lo logramos mediante un sistema de locks con el siguiente comportamiento:

1. El hilo principal crea y adquiere (bloquea) todas las locks.
2. El hilo de cada controlador de CPU intenta adquirir su lock correspondiente.
3. Cuando se crea un controlador de CPU, el hilo principal desbloquea la lock correspondiente a ese controlador.

La barrera sigue siendo necesaria para esperar a que el hilo de tareas de host cree la cola de tareas de host.

Esta fue la solución adoptada inicialmente. Pero más adelante se decidió usar un fichero de configuración y crear todos los controladores dentro del `__ctrl_block__`, Como se discute en la sección 3.4. Esto permite simplificar y hasta eliminar estas sincronizaciones.

Como los controladores ahora se crean antes que los hilos, los hilos de los controladores no tienen que esperar a que su controlador se cree. Y la barrera, que ahora solo sirve para esperar a que el hilo de tareas de host cree la cola de tareas de host, se puede eliminar si movemos la creación de esa cola al mismo sitio en el que se crean los controladores.

Asignación de hilos

Otro problema relacionado es como sabe cada hilo cuál es su controlador. En la lista global de controladores están todos los controladores por orden de creación.

Una forma de resolver esto es crear una lista de tamaño igual al número de controladores de CPU, donde cada valor es el índice global de ese controlador. Podemos guardar los valores en esta lista a medida que se vayan creando los controladores. Después cada hilo puede usar su índice de OpenMP (restando los hilos de gestión) para acceder a esa lista y mirar cuál es su controlador correspondiente.

Otra opción que no requiere del uso de variables globales es que cada controlador recorra

la lista de controladores llevando la cuenta de cuántos controladores de tipo CPU encuentra. Una vez que encuentre un número de controladores de CPU igual a su índice de OpenMP menos el número de hilos de gestión (actualmente dos, el principal y el de tareas de host) habrá encontrado su controlador.

Inicialmente, se implementó la primera solución y posteriormente se cambió por la segunda.

4.1.3 Cambios de contexto de CUDA

En el backend de CUDA de Controller se usa la API del runtime de CUDA, ya que es más simple programar con ella.

Todas las llamadas a la API se hacen desde el hilo principal, por lo tanto, al usar múltiples controladores debemos tener cuidado con el contexto de CUDA activo en cada momento.

La API del runtime de CUDA, a diferencia de la API del driver, provee una gestión implícita de los contextos [40]. La API de runtime decide por sí misma qué contexto usar para un hilo: si un contexto se ha seleccionado para el hilo mediante la API del driver, el runtime lo usará, pero si no existe tal contexto, usa un "contexto primario". Los contextos primarios se crean según sea necesario, uno por dispositivo por proceso y usan un sistema de conteo de referencias para destruirlos una vez no queden referencias a ellos. Dentro de un proceso, todos los usuarios de la API de runtime compartirán el contexto primario, a menos que se haya seleccionado un contexto para cada hilo. El contexto que utiliza el runtime, es decir, el contexto actual o el contexto primario, puede sincronizarse con `cudaDeviceSynchronize` y destruirse con `cudaDeviceReset`.

Para cambiar el contexto activo en un hilo, podemos usar la función de la API de runtime `cudaSetDevice`, esto es necesario para las siguientes operaciones:

- Reservar memoria en el dispositivo.
- Lanzar un kernel a un stream.
- Crear un stream.
- Crear un evento de CUDA.

4.1.4 Compilación de kernels de OpenCL

Como ya mencionamos en la sección 3.3, la compilación de kernels de OpenCL requiere de algunas modificaciones.

En OpenCL, los kernels se pueden compilar en runtime a partir de una cadena de caracteres o cargar un binario ya compilado. Este proceso es parecido aunque ligeramente diferente

en los backends de OpenCL GPU y FPGA. En ambos se realiza la carga/compilación de todos los kernels antes de que el controlador pueda utilizarse.

Sistema usado hasta ahora

En el backend de OpenCL GPU, los kernels se compilan en runtime a partir de cadenas de caracteres. Este proceso consta de tres pasos:

1. En tiempo de compilación:
   (a) Cuando se define un kernel con el macro CTRL_KERNEL, se definen varias variables globales que llevan en el nombre de la variable el nombre del kernel, su tipo y subtipo. Estas variables contienen respectivamente: el cuerpo del kernel (sin la cabecera), el nombre del kernel, el string del kernel con la cabecera, puntero al programa (`clProgram`) y puntero al kernel (`clKernel`).
   (b) En el macro CTRL_KERNEL_PROTO, se define una función "constructora". Las funciones de este tipo se marcan como `__attribute__((constructor))` y se ejecutan antes del main.
2. Antes del main: la función mencionada en el punto anterior se ocupará de:
   - Reservar memoria para los punteros del programa y del kernel.
   - Crear el string del kernel completo a partir de los argumentos del macro y el string con el cuerpo del kernel.
   - Guardar los punteros y el string completo en un nuevo nodo de una lista enlazada que contendrá la información de todos los kernels.
3. En la creación del controlador: se recorre la lista enlazada con la información de todos los kernels compilándolos para el dispositivo de este controlador.

Hecho esto, cuando se lanza un kernel, se usan las variables globales mencionadas anteriormente con el nombre, tipo y subtipo correspondientes.

En el backend de FPGA, los kernels se compilan aparte usando un compilador diferente, por lo tanto, los kernels de este backend deben ser cargados a partir de estos ficheros binarios. El proceso es similar, la diferencia es que en vez de trabajar con *strings* del código de kernel se trabaja con un *path* a un bichero binario que se lee para cargar el kernel durante la creación del controlador.

Nuevo sistema

Con el nuevo modelo de múltiples controladores es necesario que los punteros al programa y al kernel sean listas de tamaño número de controladores de OpenCL GPU y FPGA

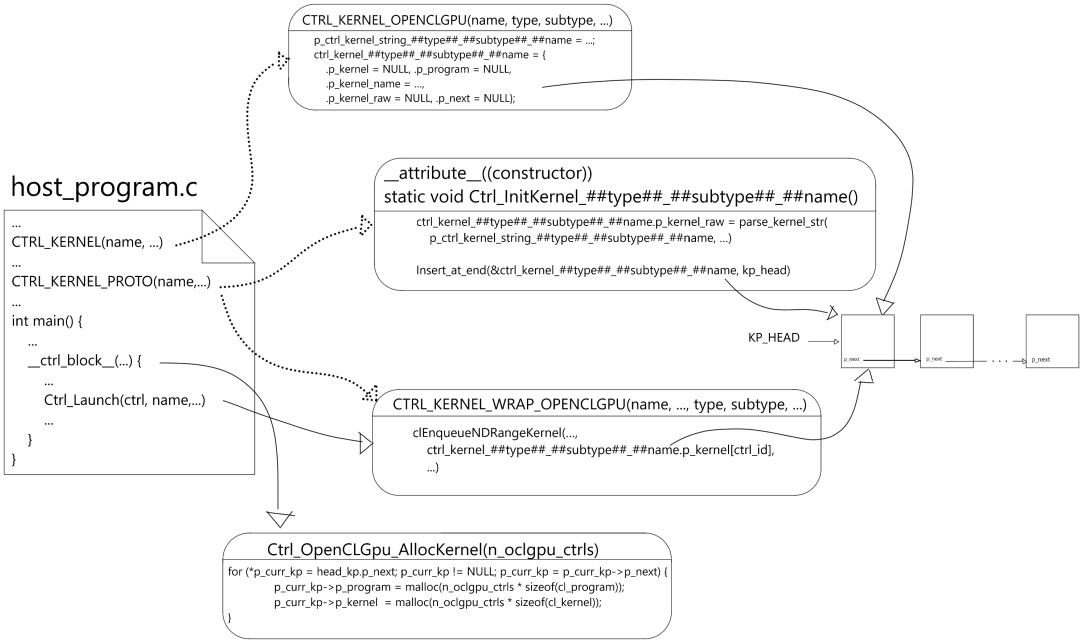

Figura 4.1: Diagrama que muestra la compilación de kernels de OpenCL GPU en Controller. Las cajas con esquinas redondas muestran funciones (en minúscula) o macros (en mayúsculas). Las flechas discontinuas del programa host representan las expansiones del preprocesador, mientras que las sólidas muestran la ejecución del código

respectivamente. El problema es que la memoria para estos punteros se reserva en la función constructora que se ejecuta antes del main y en ese momento aún no sabemos cuantos controladores se van a utilizar.

En el diagrama 4.1 podemos ver una representación del nuevo sistema. El macro CTRL_KERNEL define una variable que es un nodo de la lista enlazada con los punteros y strings necesarios en su interior, en la función constructora se enlazan estos nodos, en el `__ctrl_block__` se reserva memoria para todos los kernels iterando sobre la lista enlazada y finalmente cuando se crea cada controlador se itera sobre la lista compilando los kernels.

Cuando se lanza un kernel, se usan los datos de la estructura apuntada por la variable global con tipo, subtipo y nombre correspondientes.

Un pequeño detalle adicional es que cada controlador debe tener un ID dentro de los controladores de su mismo tipo para usarlo como índice a la hora de acceder a su versión compilada de un kernel concreto.

4.1.5 Creación de controladores

Como se explica en la sección 3.4, se ha decidido mover la creación de los controladores al interior del macro `ctrl_block`. La información necesaria para la creación de los controladores se suministra mediante un fichero de configuración que se parsea dentro del macro.

En el listing 4.1 podemos ver el formato de este fichero de configuración. En este archivo, cada controlador se define en una línea que comienza con el tipo de controlador, seguido de los parámetros necesarios para su funcionamiento. Es posible especificar configuraciones para varias máquinas en el mismo fichero, y las descripciones de los controladores para cada máquina se encuentran después de una línea que comienza con "NODE", seguida del nombre de la máquina correspondiente.

```
1  NODE <node_name> [host_numa_node]
2  CUDA <device_id>
3  OpenCL_GPU <platform_id> <device_id>
4  CPU <n_threads> <device_numa_start>-<device_numa_end> <transfers>
5  FPGA <platform_id> <device_id> <exec_mode>
```

Listing 4.1: Formato del fichero de configuración de Controller. Cada linea después de NODE representa un dispositivo diferente. Los parámetros entre corchetes son opcionales.

De esta forma, el usuario ya no necesita llamar a la función para crear controladores y en su lugar usará la función `Ctrl_Get(i)` para obtener el i-ésimo controlador definido en el fichero de configuración.

También se ha movido la destrucción de los controladores a un punto común, la función `Ctrl_EndBlock` que deberá ser llamada antes de acabar el ctrl_block.

4.2 Tiles compartidas entre varios controladores

El siguiente paso es que los diferentes controladores puedan interaccionar entre sí. Esto sucederá, por ejemplo, cuando tengamos una estructura de tipo tile asociada a varios controladores simultáneamente.

4.2.1 Interacciones entre controladores al lanzar una tarea

Cuando una tile está asociada a múltiples controladores de forma simultánea, las operaciones que se realizan sobre esta tile deben sincronizarse correctamente. Esto implica que debemos realizar esperas adicionales a operaciones de otros controladores. En la tabla 4.1 se pueden ver las reglas de dependencias actualizadas. Con las marcas en blanco señalizando

esperas a operaciones del mismo controlador y las azules marcando esperas a operaciones de todos los controladores asociados a la tile.

Op \ Waits to	K(In x)	K(Out x)	H(In x)	H(Out x)	HTD(x)	DTH(x)
K(In x)		x			x	
K(Out x)	x	x			x	x
H(In x)				x		x
H(Out x)			x	x	x	x
HTD(x)	x	x		x		x
DTH(x)		x	x	x	x	
Free(x)	x	x	x	x	x	x
Wait(x)	x	x	x	x	x	x

Cuadro 4.1: Dependencias entre operaciones del nuevo modelo Controller aplicadas sobre una tile *x*. En las operaciones de kernel (*K*) y tareas de host (*H*) distinguimos entre los casos en los que *x* es un parámetro de entrada (In) o salida (Out). Una marca negra en la celda significa que la operación de esa fila deberá esperar a la operación de la columna correspondiente del mismo controlador. Una marca en rojo indica que deberá esperar a esa operación de todos los controladores asociados a la tile *x*.

Tareas de host

Como hemos visto, las tareas de host son un caso en el que se debe esperar a eventos de otros controladores. Pero ese no es el único cambio que hay que realizar.

Hasta ahora las tareas de host se lanzaban asociadas a un controlador y de forma ligeramente diferente en cada backend:

- En CUDA se usaban eventos de CUDA así como la función `cudaLaunchHostFunc` para poder sincronizar con los eventos de CUDA.
- En OpenCL GPU y FPGA se lanzaba a una cola de host usando eventos genéricos de tipo OpenCL para sincronizar. Para que el resto de operaciones de OpenCL pudieran esperar a estas tareas de host, se usaba un evento de usuario de OpenCL que se señalaba mediante una tarea introducida en la cola de host justo después de la tarea de host.
- En CPU se lanzaba a una cola de host y se usaban eventos de CPU para sincronizar.

Ahora las tareas de host pueden contener tiles asociadas a múltiples controladores, por lo tanto, las tareas de host no deben ir asociadas a un controlador. Asimismo, es necesario que se lancen desde un punto común. Por eso se han movido de los backends a la parte del core, para cada tile se espera a todos los eventos apropiados de los controladores a los que esté asociada usando eventos genéricos.

4.2.2 Sistema de eventos

Para que los controladores puedan sincronizarse entre sí, es necesario que haya interoperabilidad entre sus eventos. Esto ya se ha mencionado en la sección 3.2. Utilizaremos la sincronización de eventos en el host explicada en la sección 3.2.3. A continuación se detalla como se ha realizado la implementación de esta solución.

Hilo gestor de colas

El hilo gestor de colas se ocupa de la gestión de las colas de host de los controladores. En el listing 4.2 se puede ver el código que ejecuta este hilo. Como podemos ver, tiene una lista de colas de host sobre las que va iterando extrayendo la última tarea disponible de cada una.

Como vemos en la línea 34, si la tarea es una espera a un evento, hará "test" del estado del evento, si el evento está completado eliminará esta tarea de la cola y pasará a la siguiente, si el evento aún no se ha completado, simplemente lo dejará en su sitio y pasará a la siguiente cola.

Como vemos en la línea 22, si la tarea es una operación (kernel, HTD o DTH), eliminará la tarea de la cola, llamará a la función del backend correspondiente para encolarla en la cola del driver apropiada (indicada en la propia tarea) y actualizará el evento genérico correspondiente al segundo periodo de la operación (también indicado en la tarea).

Cabe mencionar, que en la versión anterior de Controller se usaba una cola por tile para las transferencias de memoria de esa tile. En esta nueva versión, cada controlador usa dos colas para las transferencias de memoria, una para las operaciones `MoveTo` y otra para las operaciones `MoveFrom`. De esta forma, cada controlador tiene una cantidad fija de colas y, por lo tanto, la lista de colas que maneje el hilo gestor de colas tiene un tamaño fijo.

Detalles de los eventos de CUDA

En el caso de CUDA, los eventos se crean con `cudaEventCreate`, se almacena información sobre la última tarea encolada en un stream en el evento con `cudaEventRecord`, se puede hacer "test" con `cudaEventQuery` y se destruyen con `cudaEventDestroy`.

En CUDA la forma normal de trabajar con eventos en los streams de CUDA es:

1. Crear el evento mediante `cudaEventCreate`.
2. Almacenar información en el evento con `cudaEventRecord`.
3. Encolar una espera al evento en un stream de CUDA.
4. Reutilizar el evento con otra llamada a `cudaEventRecord`. Esto no afecta a la espera encolada en el paso anterior.

```
void Ctrl_Thread_QueueManager() {
    int n_queues = 0;
    for (int i = 0; i < Ctrl_GetNCtrls(); i++) {
        n_queues += Ctrl_GetNumQueues(&p_ctrl_global[i]);
    }
    Ctrl_TaskQueue  *pp_queues[n_queues];
    // obtener las colas de host de cada controlador
    Ctrl_TaskQueue **pp_aux = pp_queues;
    for (int i = 0; i < Ctrl_GetNCtrls(); i++) {
        pp_aux = Ctrl_GetQueues(&p_ctrl_global[i], pp_aux);
    }

    while (true) {
        for (int i = 0; i < n_queues; i++) {
            // no hace pop, la tarea extraida sigue en la cola
            Ctrl_Task *p_task = Ctrl_TaskQueue_GetNext(pp_queues[i]);
            if (p_task == NULL) {
                // no hay tareas en esta cola, pasar a la siguiente
                continue;
            }
        switch (p_task->task_type) {
            case CTRL_TASK_TYPE_KERNEL:
            case CTRL_TASK_TYPE_MOVEFROM:
            case CTRL_TASK_TYPE_MOVETO:
                // eliminar la tarea de la cola
                Ctrl_TaskQueue_Pop(pp_queues[i]);
                // ctrl_type dentro de p_task determina el backend
                Ctrl_<arch>_ExecTask(p_task);
                // last_finished se usa para los eventos de cpu
                #pragma omp atomic update
                pp_queues[i]->last_finished++;
                Ctrl_TaskQueue_FreeTask(p_task);
                break;
            case CTRL_TASK_TYPE_WAITEVENT:
                if (Ctrl_GenericEvent_Test(p_task->event)) {
                    // el evento está completado
                    // eliminar tarea de la cola y liberar
                    Ctrl_TaskQueue_Pop(pp_queues[i]);
                    Ctrl_GenericEvent_Release(p_task->event);
                    #pragma omp atomic update
                    pp_queues[i]->last_finished++;
                }
                break;
            case CTRL_TASK_TYPE_DESTROYCTRL:
                // punto de salida
                for (int j = 0; j < n_queues; j++) {
                    Ctrl_TaskQueue_Destroy(pp_queues[j]);
                }
                return;
            }
        }
    }
}
```

Listing 4.2: Codigo del hilo gestor de colas.

No obstante, cuando se trabaja con estos eventos en el host, el comportamiento es diferente. Si se crea una copia de un evento (ej. para introducir una espera en una cola) y posteriormente se intenta llamar a `cudaEventRecord` la copia realizada anteriormente se ve afectada.

Por lo tanto, en nuestro modelo, crearemos un nuevo evento para cada operación que se introduzca en la cola, encolar esperas a estos eventos funciona igual que hasta ahora, salvo que utilizaremos la interfaz de los eventos genéricos, y el "record" será realizado por el hilo gestor de colas inmediatamente después de introducir la tarea en la cola del driver.

Otro problema llega a la hora de destruir los eventos. No podemos destruirlos hasta que sepamos que ya no queda ninguna copia del evento disponible. La solución es añadir a los eventos genéricos un sistema simple de conteo de referencias. Este contador tendrá el siguiente comportamiento:

- Debe mantenerse entre copias del evento genérico. Con un puntero, por ejemplo.
- Comenzará a 1 cuando el evento se cree.
- cada vez que se cree una copia del evento el contador aumentará (ej. al encolar una espera al evento).
- Cada vez que desaparezca una copia del evento el contador se decrementará (ej. al completar una espera al evento).
- Si el contador llega a 0 el evento será destruido.

Este sistema es similar al sistema de conteo de referencias que usan los eventos (y otros objetos) de OpenCL.

Detalles de los eventos de OpenCL

En el caso de OpenCL, los eventos son devueltos por las operaciones que se encolan en sus colas del driver. Las funciones como `clEnqueueNDRangeKernel` (encolar un kernel) tienen un parámetro de tipo `cl_event*` que representará una espera a esa operación. Estos eventos no pueden ser modificados (no existe una función "record" en OpenCL). Se puede hacer "test" usando la función `clGetEventInfo` y como se ha mencionado previamente, se puede controlar cuando se destruyen mediante su sistema de conteo de referencias.

Hay dos problemas con estos eventos.

Son inmutables: Los eventos genéricos de tipo OpenCL actualmente usan un evento de OpenCL (`cl_event`). Estos eventos no son modificables, y como hemos dicho previamente, los eventos del driver no se crean hasta que la tarea es procesada y se introduce en la cola del

driver y, por lo tanto, es necesario que sea posible actualizar las tareas de espera que usen ese evento. La solución es cambiar el comportamiento de los eventos genéricos de tipo OpenCL usando un puntero a un evento de OpenCL en vez de un evento como se hace ahora. La memoria para este evento se reservará cuando se cree el evento genérico y se liberará cuando el evento sea destruido.

Sin evento no hay conteo de referencias: Como hemos mencionado al hablar sobre el conteo de referencias en eventos de CUDA, esta referencia se tiene que incrementar cuando se encola una espera al evento, pero cuando esto sucede es posible que la tarea aún no haya sido procesada por el hilo gestor de colas y, por lo tanto, el evento de OpenCL aún no se haya creado. Por lo tanto, en este punto aún no podemos usar el sistema de conteo de referencias de los eventos de OpenCL y debemos, por lo tanto, usar el mismo sistema que en los eventos genéricos de CUDA.

4.2.3 Cambios en las tiles

Las estructuras de metadatos que Controller utiliza para expandir la funcionalidad de las HitTiles de Hitmap necesitan modificaciones para permitir que sea posible asociar una tile con múltiples controladores simultáneamente.

Estructura Ctrl_Tile

Esta estructura es la extensión de las HitTiles que las permiten funcionar con el modelo de Controller. Es una estructura que cuelga del campo `ext` de la HitTile. Es en esta estructura donde se almacenan los eventos que permiten sincronizar las operaciones que se han de realizar sobre la tile, así como, el puntero a la imagen de memoria de la tile en el dispositivo asociado al controlador y cualquier otra información necesaria.

Hasta ahora, cada backend utilizaba su propia estructura de tipo `Ctrl_<arch>_Tile` que colgaba del campo *ext* mencionado anteriormente y contenía toda la información de la tile relacionada con el dispositivo.

Ahora que una tile debe poder estar asociada con varios controladores al mismo tiempo debemos encontrar una nueva solución. Como podemos ver en el diagrama 4.2, ahora del campo `ext` de una HitTile cuelga una estructura de tipo `Ctrl_Tile` que contiene información de la tile que es común a todos los dispositivos (eventos del host, estado de la imagen del host...) así como una lista de estructuras de tipo `Ctrl_Tile_Impl`.

La estructura `Ctrl_Tile_Impl` contiene los datos de la tile relevantes a un controlador concreto. De ella "heredan" los tipos de ctrl tile de cada backend. Esta "herencia" está implementada usando una *union* con los tipos de ctrl tiles existentes previamente.

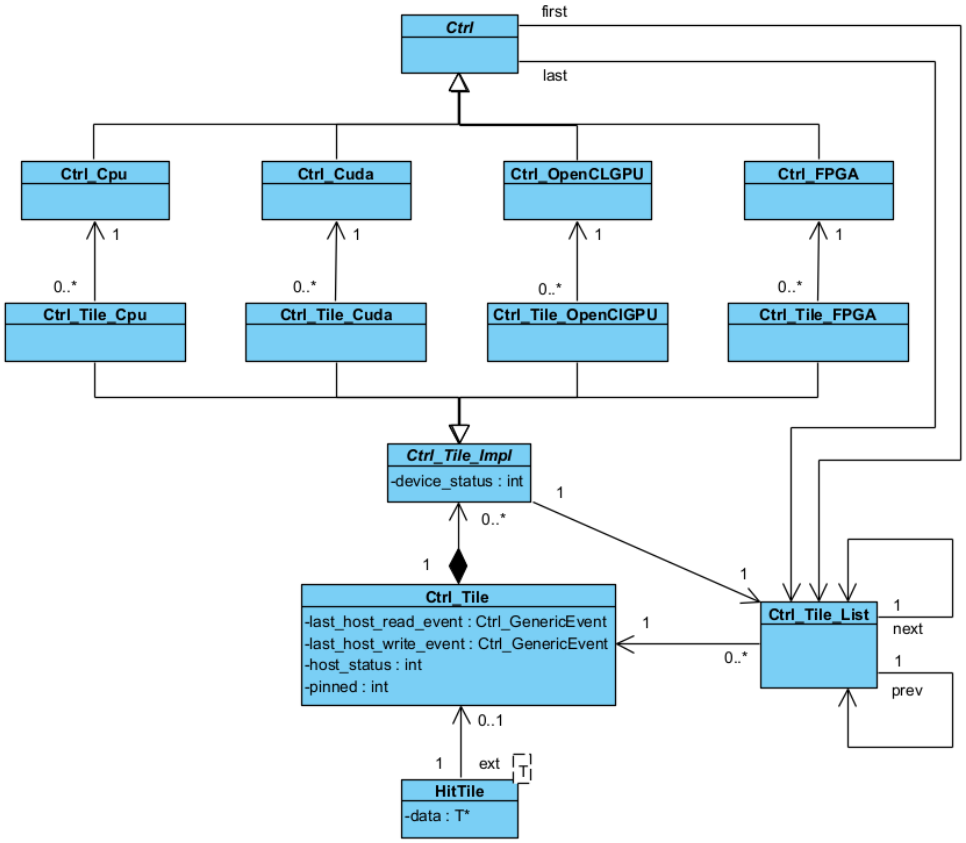

Figura 4.2: Diagrama de clases del nuevo diseño de Ctrl Tile.

Estructura Ctrl_Tile_List

Esta estructura representa una lista enlazada de las tiles asociadas a un controlador. Se utiliza para las operaciones que requieren esperar a todas las tiles de un controlador. Hasta ahora, estaba definido en cada backend con un nombre diferente (`Ctrl_<arch>_Tile_List`), ya que, contenía un puntero a la estructura ctrl_tile de ese backend.

Ahora esto ya no es necesario que se defina en cada backend porque un nodo de esta lista apuntará a una estructura `Ctrl_Tile`.

Creación de tiles

La función `Ctrl_Domain`, usada para crear una tile, ahora crea una tile vacía que no está asociada a ningún controlador (inicializa la HitTile y la estructura `Ctrl_Tile`). Por lo tanto ya no recibe un controlador como parámetro y se ha movido al core de Controller.

```
HitShape shape = hitShapeSize(SIZE, SIZE)

// crea la tile (sin memoria reservada)
HitTile_float m1 = Ctrl_Domain(float, shape);
// reserva memoria en el host y el dispositivo de ctrl1
Ctrl_Alloc(ctrl1, m1)
// reserva memoria en el dispositivo de ctrl2
Ctrl_Alloc(ctrl2, m1)
// libera memoria del dispositivo de ctrl1
Ctrl_Free(ctrl1, m1)
// libera memoria del dispositivo de ctrl2 y de host
Ctrl_Free(ctrl2, m1)

// version alternativa usando el atajo DomainAlloc
// crear la tile y reservar memoria en host y dispositivo de ctrl1
HitTile_float m2 = Ctrl_DomainAlloc(ctrl2, float, shape);
// reservar memoria en dispositivo de ctrl2
Ctrl_Alloc(ctrl1, m2)
```

Listing 4.3: Ejemplo de como asociar una tile a varios controladores.

Reserva de memoria en las tiles

La función `Ctrl_Alloc`, usada para reservar memoria en el host y/o en el dispositivo asociado al controlador, es el mecanismo utilizado para asociar una tile con un controlador. La semántica es la siguiente:

- Cuando se llama con una tile y controlador que no están asociados, se inicializan las estructuras de la tile referentes a este controlador.
- Cuando se llama con una tile que ya tiene memoria reservada en el host, solo se reserva memoria en el espacio de memoria del dispositivo.

En el listing 4.3, podemos ver un ejemplo de uso de las funciones `Ctrl_Domain` y `Ctrl_Alloc` para crear y reservar memoria para una tile 2D de floats asociada a dos controladores. También podemos ver una versión alternativa más compacta usando la función `Ctrl_DomainAlloc`.

Liberación de memoria de las tiles

De forma similar a la reserva de memoria, ahora para liberar la memoria de una tile que está asociada a varios controladores es necesario llamar a `Ctrl_Free` con cada uno de los controladores asociados a la tile, la tile solo será destruida y la memoria de host liberada una vez que se llame a esta función con el último controlador que esté asociado a la tile. En el listing 4.3 podemos ver un ejemplo.

4.2.4 Sobre la reserva de la memoria del host

Una consideración importante a la hora de reservar la imagen de memoria del host de una tile es el uso de formas especiales de reservar memoria, tales como memoria "pinned" para optimizar las transferencias con el dispositivo o reservar la memoria alineada de una cierta forma.

Diferentes dispositivos prefieren (o requieren) tener la memoria reservada de diferentes formas. Hasta ahora esto era relativamente simple, cada tile está asociada a un solo controlador, así que simplemente hay que reservar memoria de la mejor forma posible para ese dispositivo. Pero ahora, si una tile está asociada a varios dispositivos, es posible que acabe asociada a varios dispositivos con formas diferentes e incompatibles.

Memoria pinned en CUDA

En CUDA para optimizar las transferencias de memoria se recomienda usar memoria pinned [41], esto consiste en usar memoria cuyas páginas no puedan ser movidas a la swap, garantizando de esta forma que la dirección física de esta no cambia y permitiendo al driver usar DMA para realizar las transferencias.

Para reservar memoria pinned en CUDA se pueden usar las funciones `cudaMallocHost` o `cudaHostAlloc` siendo la segunda una versión de la primera con un parámetro extra para controlar su comportamiento. Para liberar esta memoria se usa la función `cudaFreeHost`.

También es posible usar la función `cudaHostRegister` para convertir memoria reservada previamente por otros medios en pinned. Antes de liberar esta memoria debe ser "dada de baja" mediante la función `cudaHostUnregister`.

Por defecto, la memoria pinned solo se considera pinned para el dispositivo activo, pero podemos cambiar este comportamiento mediante las flags `cudaHostAllocPortable` y `cudaHostRegisterPortable`.

Memoria pinned en OpenCL GPU

En OpenCL no hay forma de garantizar que la reserva de memoria sea pinned, no obstante en las guías de mejores prácticas de Nvidia [42] y AMD [43] se menciona un mecanismo para darle una pista al runtime para que reserve la memoria como pinned. El proceso se puede ver en el listing 4.4.

Memoria alineada en FPGA

En cuanto al backend de FPGA, es preferible tener la memoria alineada a 64 bytes para lograr transferencias más rápidas gracias a DMA. El bus de memoria de las FPGAs de Intel

```
// alloc
cl_mem pinned_buf = clCreateBuffer(context, CL_MEM_ALLOC_HOST_PTR, SIZE,
    NULL, NULL);
void *p_data = clEnqueueMapBuffer(queue, pinned_buf, CL_TRUE, CL_MAP_READ |
    CL_MAP_WRITE, 0, SIZE, 0, NULL, NULL, NULL);
...
// free
clEnqueueUnmapMemObject(queue, pinned_buf, p_data, 0, NULL, NULL);
clFlush(queue);
clFinish(queue);
clReleaseMemObject(pinned_buf);
```

Listing 4.4: Ejemplo de reserva de memoria pinned en OpenCL.

tiene 64 bytes, es decir, los datos se transfieren en ráfagas de ese tamaño.

Solución final

Para este prototipo, usaremos la flag `cudaHostAllocPortable` en el backend de CUDA, ya que no perece tener ningún efecto negativo, y dejar que el orden de llamadas a `Ctrl_Alloc` defina que controlador se ocupa de reservar la memoria.

4.3 Movimientos de memoria implícitos entre controladores

El último detalle es que la semántica de transferencias implícitas del modelo Controller funcione con múltiples dispositivos al mismo tiempo. De manera similar al modelo previo de Controller, esto consiste en analizar el estado de la memoria de la tile e introducir las transferencias necesarias.

4.3.1 Tareas de host

Cuando se lanza una tarea de host, esta trabaja sobre la imagen de host de una tile, por lo tanto, si la imagen del host está en estado inválido debemos realizar un movimiento de memoria implícito.

Ahora que la tile puede estar asociada a varios controladores, debemos comprobar cuál de ellos contiene la tile actualizada y encolar una operación `MoveFrom` a ese controlador.

Si la tile no estuviera en estado válido en el dispositivo de ninguno de los controladores asociados, se lanza un aviso y se procede a usar la memoria del host sin realizar ninguna transferencia.

4.3.2 Kernels

Cuando solo había un controlador asociado a cada tile, al lanzar un kernel, si la imagen del dispositivo de la tile está en estado inválido, se realizaba una operación `MoveTo` para transferir la imagen del host al dispositivo. Si la imagen de host tampoco estaba en estado válido, se lanzaba un aviso.

Ahora ese sistema sigue igual, salvo que en el caso de que la imagen de host esté también en estado inválido, se comprueba el estado de la tile en los otros controladores asociados y si se encuentra uno con estado válido, se encola primero una operación `MoveForm` a ese controlador para mover la tile al host y luego una operación `MoveTo` en el controlador que estaba lanzando el kernel.

Kernels en CPU con 0 copy

En CPU, cuando el modo 0 copy está activado es un caso especial. En este caso, el controlador utiliza la imagen de host de esa tile para los kernels, por lo tanto, el comportamiento es el mismo que una tarea de host. En el caso de estar la tile inválida en el host, se busca un controlador asociado a esa tile que tenga la imagen válida y se usa una operación `MoveFrom` sobre ese controlador.

4.4 Resumen

En este capítulo se han repasado los problemas afrontados y las soluciones escogidas para la implementación de la solución propuesta en el contexto de la anterior implementación del modelo Controller. El diseño y aplicación de las soluciones ha sido incremental. El resultado final cumple con todas las funcionalidades necesarias para utilizar controladores con varios dispositivos heterogéneos en el mismo proceso, con movimientos implícitos de datos a través de las tiles compartidas entre controladores. La eficiencia de la solución se verificará experimentalmente en el siguiente capítulo.

Capítulo 5

Estudio experimental

En este capítulo se detalla el estudio experimental realizado, incluyendo los objetivos, casos de estudio y resultados del mismo.

5.1 Metodología

5.1.1 Objetivos del estudio

El objetivo de esta experimentación es validar el modelo propuesto, comprobar su eficiencia y observar las diferencias de comportamiento con respecto a la versión previa.

5.1.2 Plataforma de ejecución

El estudio experimental se ha llevado a cabo en la máquina *manticore* del clúster de investigación del grupo Trasgo, con las siguientes características:

- Procesador: 2x Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10 GHz
- Nodos NUMA: 2 nodos NUMA con 24 núcleos con hyperthreading.
- Memoria RAM total: 64 GB GDDR3.
- 2x NVIDIA Tesla V100 PCIe 32 GB GPU.
- 2x AMD Vega 10 XT Radeon PRO WX 9100 GPU.

El sistema operativo de *manticore* es la distribución de Linux CentOS versión 7.9.2009. Todos los programas se compilan con GCC v10.3 y se lanzan desde un frontend usando slurm. La versión de CUDA instalada es 11.3 y la de ROCm es 4.2.0. Las librerías de OpenCL usadas son las incluidas en las instalaciones correspondientes de CUDA y ROCm.

En el caso de las aplicaciones de CPU, se ha usado la versión de la librería de Intel (liomp5)

en vez de libgomp debido a un problema de rendimiento que tiene esta última relacionado con la creación de los hilos en regiones paralelas anidadas. La versión de icc instalada es 2021.5.0.

5.1.3 Escenario de la experimentación

Para comprobar el rendimiento del nuevo modelo de sincronización con respecto a la versión previa, usaremos tres aplicaciones, una de ellas con 4 variantes. Son representativas de las diferentes situaciones que puedan surgir, incluyendo escenarios limitados por memoria, computo, comunicación y entrada-salida. Estas aplicaciones son: *Hotspot*, *Matrix Pow* y *Sobel YUV* y se explican en detalle más adelante.

Los programas para cada caso de estudio se desarrollan en CUDA, OpenCL, OpenMP y Controller. Desarrollamos dos versiones de los programas CUDA de referencia y dos versiones de los programas OpenCL de referencia: una síncrona simple (Ref. Sync) y una versión optimizada manualmente con la mejor combinación posible de operaciones asíncronas (Ref. Async). Para OpenMP solo podemos contar con una versión síncrona (Ref. Sync) sin desarrollar un código con un diseño completamente diferente basado en tareas. En cuanto a Controller las versiones síncrona y síncrona salen del mismo código, ya que la política se puede controlar en tiempo de ejecución. Las versiones que usan el modelo anterior de Controller se marcan como *Base* y las que usan el nuevo como *Multi*. Para las versiones de CPU de Controller, adicionalmente se ha probado el modo normal y el de 0 copy, activando (Copy) y desactivando (No copy) las transferencias de memoria.

Adicionalmente, se ha diseñado e implementado una aplicación que pueda mostrar las ventajas del uso de varios dispositivos para mostrar las nuevas funcionalidades de Controller. Esta aplicación es *Chain matrix multiplication*.

Para esta aplicación se han desarrollado versiones en CUDA y OpenCL usando una sola GPU y una versión de Controller que utiliza las 4 GPUs de *Manticore*, mezclando dispositivos de NVIDIA y AMD de diferentes capacidades.

Para cada experimento se ejecutan 30 repeticiones, ya que, este es un tamaño de muestra suficientemente grande para que el Teorema Central del Límite se considere aplicable. Los resultados presentados son la media de los tiempos de ejecución tras haber eliminado los outliers, es decir, aquellos resultados por debajo o por encima de la media $\pm 1.5 \times IQR$ (rango intercuartílico)

Rodinia: Hotspot stencil computation

El programa base está incluido en la suite de aplicaciones de referencia de Rodinia [44], computa el punto de estabilidad de la Ecuación Diferencial Parcial de Poisson (PDE) de la

```
n = numero de iteraciones en la simuacion
ipc = numero de iteraciones por copia
Declaracion de matrices O y D en host y device
Declaracion del buffer C
for i in 0.. n :
    Intercambio de las matrices O y D
    llamada al kernel
    if (( i % ipc ) == 0):
        transferencia del resultado en device a la matriz D en host
        copia de D a C
```

Listing 5.1: Pseudocodigo de hotspot

difusión de calor. Utiliza el método iterativo de Jacobi para un espacio discreto bi-dimensional. Este programa stencil de cuatro puntos ejecuta un número de iteraciones fijo. En cada iteración se calcula un nuevo valor por cada celda de la matriz, utilizando la información de sus cuatro vecinos. El resultado después de cada iteración se transfiere al host, guardándolo en un buffer. Por lo tanto se podría usar para comprobar resultados parciales o crear una animación de la evolución del programa.

En este caso de estudio la carga computacional es pequeña en comparación con las transferencias.

Matrix pow

Este programa es una evolución de los programas *2mm* y *3mm* de la suite de referencia PolYBench [45] para generar una cadena de multiplicaciones de matrices de longitud arbitraria, computando $C = A^n$ mediante el siguiente método iterativo:

$$C^k = C^{k-1} \times A : k \in [2 : n]$$

$$C^1 = A$$

El kernel para la multiplicación de matrices se ha obtenido de los ejemplos del toolkit de CUDA [46] y se ha portado a OpenCL.

Los resultados parciales después de cada iteración se transfieren al host para calcular la normalización de la matriz y la guarda en otro buffer. la normalización de la matriz consiste en los siguientes pasos:

- Determinar los valores mínimos y máximos en la matriz.
- Restar el mínimo de cada elemento de la matriz, y dividir el resultado por el máximo.
- Calcular la norma de los elementos como la raíz cuadrada de la suma de cada elemento

```
Reserva de tres matrices A , B y C en host y device

for i in 0.. Potencia :
    if (( i % 2) == 0):
        Llamada al kernel ( C = A * B )
        Copia de C al host
        Matriz_host = C
        Normalizacion ( Matriz_host )
    else
        Llamada al kernel ( B = A * C )
        Copia de B al host
        Matriz_host = B
        Normalizacion ( Matriz_host )
```

Listing 5.2: Pseudocodigo de MatrixPow

```
for fotograma in 0.. num_fotogramas :
    for componente in 0..2:
        transferencia de host a device
        llamada al kernel
        transferencia de device a host
for componente in 0..2:
    carga del componente
for componente in 0..2:
    escritura del resultado
```

Listing 5.3: Pseudocodigo de sobel

al cuadrado.

- Dividir cada elemento de la matriz por la norma de los elementos.

Sobel YUV

El operador Sobel [47] procesa imágenes en escala de grises para detectar bordes. Aplica dos operadores stencil a la imagen de entrada, para obtener las derivadas en las direcciones X e Y. La magnitud del gradiente se calcula en cada celda como la distancia euclídea de las celdas correspondientes en las matrices obtenido como la salida de los filtros.

Para este estudio hemos seleccionado una versión que procesa fotogramas de forma iterativa de un vídeo en formato YUV. El programa lee el fichero de entrada fotograma a fotograma. Cada fotograma tiene tres componentes. El filtro de sobel se aplica a cada componente lanzando el mismo kernel, una vez por componente. Las imágenes resultantes se transfieren de vuelta al host para almacenarla en el fichero de vídeo de salida. Cada componente de un fotograma se lee, escribe, computa y transfiere de forma separada.

Para simular diferentes escenarios de la aplicación del filtro de sobel, hemos considerado distintos escenarios:

- Las imágenes de entrada y salida se leen y escriben directamente de ficheros.
- Las imágenes de entrada se leen de un fichero, pero las de salida se guardan en memoria.
- Las imágenes de entrada se leen de un buffer de memoria, pero las de salida se escriben a un fichero.
- Las imágenes de entrada y salida se leen y escriben de buffers de memoria.

Chain matrix multiplication

Este programa calcula el resultado de cadenas de multiplicaciones de matrices de la forma $B_i = A_i \times C1 \times C2 \times C3 \times C4$ para una serie de matrices A_i que se generan en cada iteración y calcula una norma similar a la del programa *Matrix Pow* para cada iteración. Esta aplicación pretende simular un tipo de programa en el que las matrices llegan en "streaming" y se les aplican filtros parametrizados en función de características o resultados obtenidos en las imágenes anteriores. Por simplicidad, en nuestra implementación las matrices C serán constantes y no cambiarán a lo largo del programa.

En el caso de Controller, la aplicación usará las 4 GPUs de *manticore*, poniendo cada una de las matrices C constantes en una de las GPUs y moviendo los resultados parciales entre ellas. De esta forma se forma un pipeline que permite solapar las operaciones de una iteración con las de la siguiente.

Un detalle importante a resolver para que el pipeline funcione bien y se puedan solapar las operaciones es que cada iteración tiene 2 tareas de host, la generación de la matriz A_i y el cálculo de la norma de B_i. Como Controller mantiene el orden en el que se encolan las tareas de host, habría que tener mucho cuidado con el orden en que se lanzan estas operaciones. Para evitar esto, lanzaremos la norma como un kernel de un dispositivo de CPU que se ejecutará con un solo hilo. Este sistema es a efectos prácticos como tener otra cola de tareas de host independiente.

En las aplicaciones de referencia se utiliza un solo dispositivo debido a la enorme complejidad que requeriría implementar una versión equivalente que combine GPUs de Nvidia y AMD.

Para permitir aún más solapamiento, se han usado dos sets de matrices cuyos punteros se intercambian cada iteración.

En la imagen 5.1 podemos ver una imagen del profiler visual de nvidia de la aplicación Chain matrix multiplication. Solo se muestran las partes relativas a las GPUs de Nvidia. Se puede observar como en las primeras iteraciones los kernels se solapan de forma muy efectiva y a partir de cierto punto aparecen retrasos. Esto es debido a que las GPUs de AMD instaladas en *manticore* son más lentas y no aparecen en esta imagen. Por desgracia no tenemos acceso

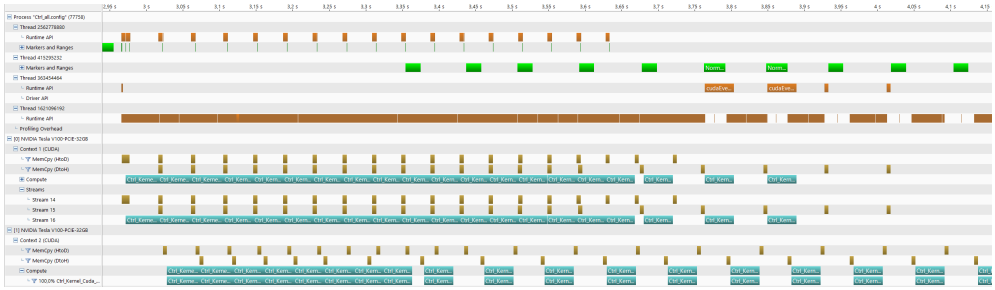

Figura 5.1: Imagen del profiling de la aplicación Chain matrix multiplication.

```
Reserva de matrices m[1-5], m_aux[1-5], c[1-4] en host y device
Init_matrix( c1, c2, c3, c4 )
HTD( c1, c2, c3, c4 ) // in ctrl to different devices

for i in 0.. Iters :
    Init_Matrix( m1 )
    HTD( m1 )
    Kernel( m2 = m1 * c1 )
    Kernel( m3 = m2 * c2 )
    Kernel( m4 = m3 * c3 )
    Kernel( m5 = m4 * c4 )
    DTH( m5 )
    norma( m5 )
    swap( m[1-5], m_aux[1-5] )
```

Listing 5.4: Pseudocodigo de Chain Matmult. En la versión de Controller cada kernel se lanza a un dispositivo y hay transferencias de memoria implicitas entre ellos.

en el cluster a un profiler que nos permita ver la parte equivalente que se ejecuta en las GPUs de AMD.

5.2 Resultados de la experimentación

En las figuras 5.2, 5.3 y 5.4 se pueden ver los gráficos de los resultados obtenidos en los casos de estudio Hotspot, Matrix Pow y Sobel YUV. Los cuadros 1 a 12 contienen la misma información con valores numéricos exactos y algunas medidas estadísticas adicionales.

Como podemos observar, en Hotspot y Matrix Pow los resultados de CUDA y OpenCL Nvidia son muy similares a los obtenidos con el modelo anterior y las versiones de referencia. Aunque si nos fijamos bien podemos ver que para los tamaños pequeños la versión mulitctrl es ligeramente más lenta. El caso extremo de esto es el caso de estudio Sobel donde al trabajar con matrices muy pequeñas y tener iteraciones muy rápidas la pérdida de rendimiento es más evidente.

Para diagnosticar la causa de esta pérdida de rendimiento analizamos la aplicación Sobel

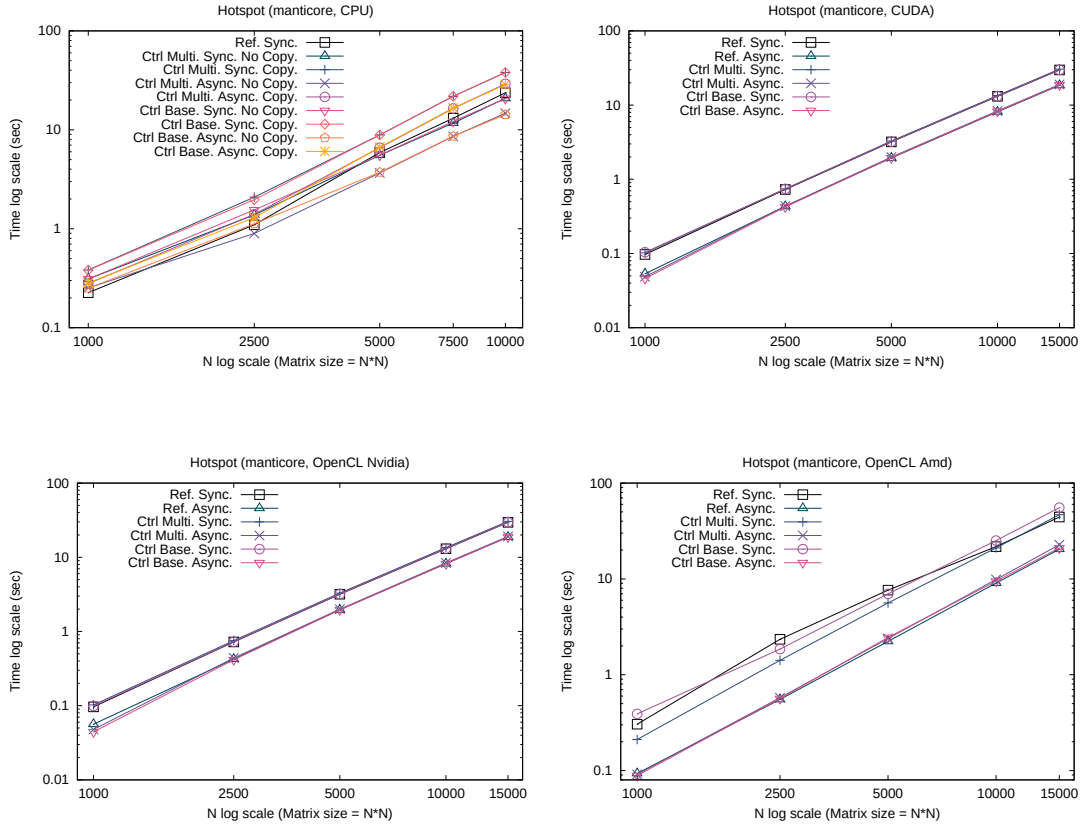

Figura 5.2: Gráficos del rendimiento de Hotspot.

con el profiler visual de Nvidia. Como podemos observar en la imagen 5.5, Sobel está limitado completamente por las tareas de host y la diferencia entre las 2 versiones es que, en la versión de multictrl, hay llamadas a funciones de eventos de CUDA (`cudaEventSynchronize` y `cudaEventDestroy`) entre una tarea de host y otra. Estas llamadas no existen en la versión original, ya que las tareas de host se sincronizaban en los streams de CUDA. Esto hace que se tarde más en pasar de una tarea a la siguiente.

Si nos fijamos en los kernels y transferencias de memoria, también podemos observar que en la versión multictrl el tiempo de lanzamiento causa retrasos (aunque en esta aplicación no se notan en el tiempo global porque están ocultos por las tareas de host). Esto es debido a que al realizar las sincronizaciones en el host, no se puede lanzar al stream de CUDA hasta que el kernel o transferencia está listo para ejecutarse.

Curiosamente en el caso de OpenCL, tanto Nvidia como AMD, observamos una perdida de rendimiento similar a la versión de CUDA en Sobel pese a que las tareas de host ya usaban nuestras colas de host de una forma similar a la actual en la versión base.

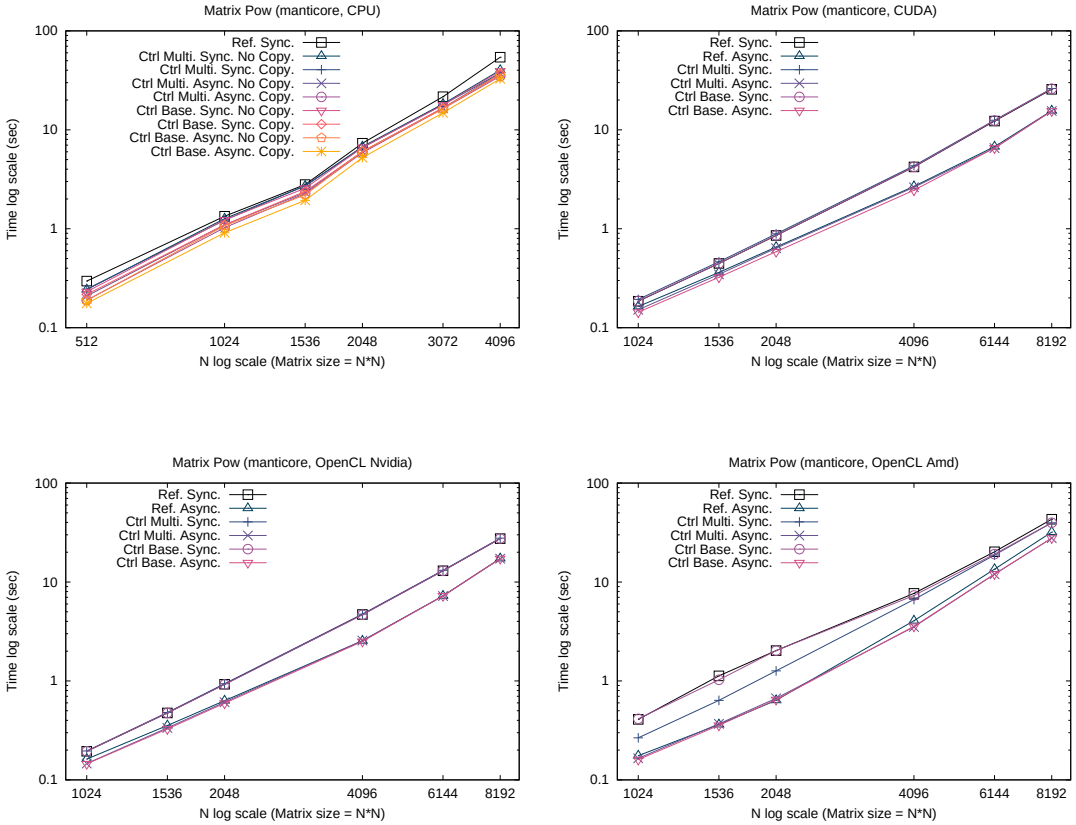

Figura 5.3: Gráficos del rendimiento de Matrix Pow.

En OpenCL AMD en Hotspot y Matrix Pow observamos una mejora significativa en el rendimiento en el modo síncrono, especialmente para matrices pequeñas. Curiosamente en Sobel YUV el comportamiento es opuesto. No estamos muy seguros de cuál es la causa de estos comportamientos y requiere un análisis más profundo con un profiler.

En cuanto a CPU, en Hotspot el comportamiento de las 2 versiones de Controller es similar. En Matrix Pow la versión multictrl es considerablemente más lenta que la versión base, no obstante ambas son más rápidas que la referencia en todas sus variantes. En Sobel YUV el comportamiento es similar, aunque en algunos casos la versión base es ligeramente más rápida que la versión multi. Esto es extraño, ya que en el backend de CPU no ha habido muchos cambios.

5.2.1 Chain matrix multiplication

En la figura 5.6 podemos ver el resultado del caso de estudio Chain matrix multiplication y en la tabla 13 podemos ver la misma información de forma numérica, así como métricas

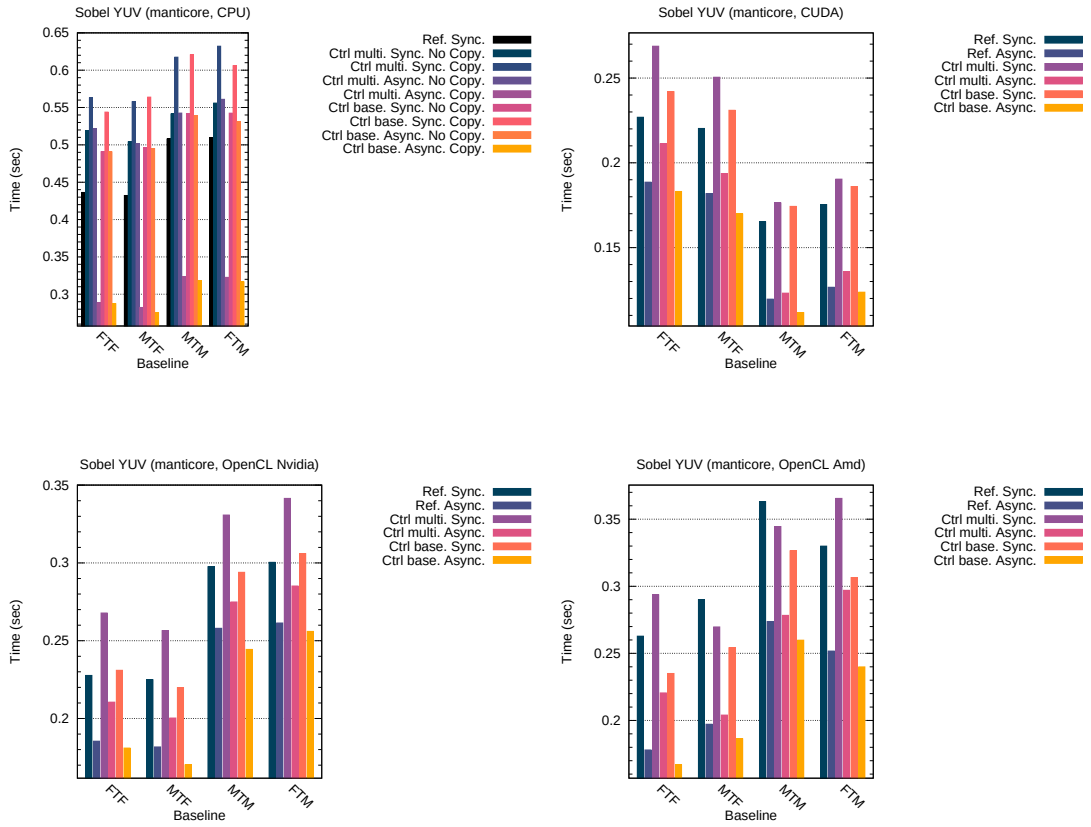

Figura 5.4: Gráficos del rendimiento de Sobel YUV

estadísticas adicionales.

Podemos observar que para tamaños de matrices muy pequeños la versión de Controller es más lenta, esto es debido a que cuanto más pequeñas son las matrices, más relevante es el efecto de las comunicaciones entre dispositivos y menos merece la pena distribuir el trabajo. Para tamaños de matrices medios y grandes, la versión de Controller es mucho más rápida gracias a poder aprovechar todos los dispositivos aceleradores de *manticore*.

5.3 Resumen

En este capítulo se ha presentado un estudio experimental para validar la solución propuesta, comparar la eficiencia de la nueva solución con respecto a la implementación anterior, y verificar las ventajas de poder utilizar varios dispositivos heterogéneos simultáneamente en un caso de uso concreto. Los resultados muestran que en algunos casos límite, la solución propuesta, dada su mayor complejidad de implementación y el tener que ser capaz de interoperar entre diversas tecnlogías, puede presentar algunas pérdidas de rendimiento. Sin embargo, en

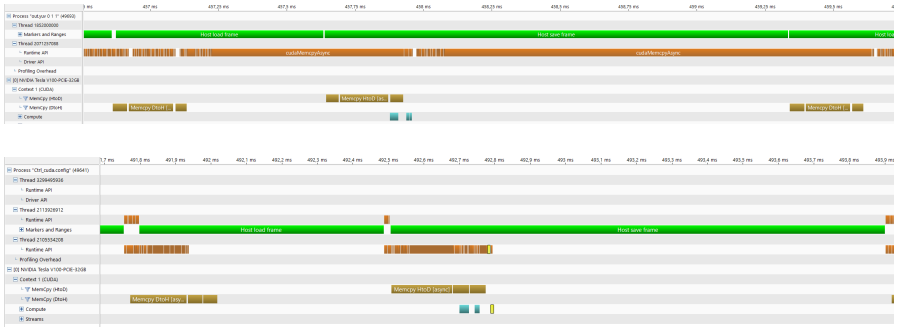

Figura 5.5: Comparativa de profiling de Sobel YUV. Arriba la versión base abajo multictrl.

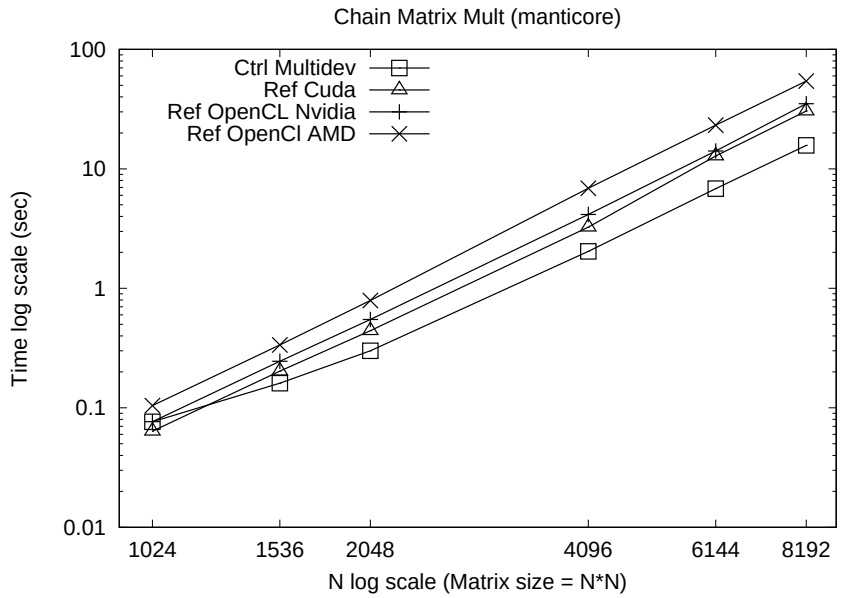

Figura 5.6: Gráfico del rendimiento de Chain matrix multiplication.

casos normales de carga estos overheads son proporcionalmente poco significativos. En el caso de la aplicación que encadena multiplicaciones de matrices a través de un pipeline entre los diferentes dispositivos de la máquina de prueba, se observa que esta solución, a pesar de los tiempos de movimiento de datos entre dispositivos, puede obtener un mejor rendimiento que intentar realizar toda la operación en uno de los dispositivos más rápidos. Esto demuestra a utilidad y eficacia de la solución propuesta.

Capítulo 6

Conclusiones

En este capítulo se revisan los objetivos cumplidos durante el desarrollo del trabajo fin de máster, las conclusiones obtenidas, así como el trabajo futuro a realizar a partir de ellas.

6.1 Objetivos cumplidos

En general, se han cumplido todos los objetivos propuestos:

1. Se ha estudiado y descrito el estado actual del modelo de programación heterogénea Controller.
2. Se han estudiado y evaluado conceptualmente otros modelos de programación paralela heterogénea.
3. Se ha propuesto una solución para, desde el mismo programa, utilizar y solapar ejecución y comunicación entre dispositivos de diferentes naturalezas o fabricantes. El nuevo modelo de Controller propuesto e implementado permite coordinar objetos controladores de diferentes arquitecturas
4. Se ha realizado un estudio experimental que ha permitido obtener información sobre el rendimiento del nuevo modelo y mostrar las nuevas funcionalidades.

6.2 Trabajo futuro

Hay ciertos detalles, optimizaciones y análisis más profundos que no se han considerado en este trabajo. Algunos de estos son:

6.2.1 Transferencias directas entre dispositivos

En este trabajo solo hemos considerado realizar las transferencias a través del host, ya que es la opción más genérica y funciona para cualquier combinación de dispositivos. No obstante, bastantes modelos como por ejemplo CUDA soportan comunicaciones directas entre dispositivos a través del bus PCIe o incluso soluciones más rápidas como nvlink.

6.2.2 Memoria pinned

El caso de la memoria pinned se menciona en la sección 4.2.4. Una tarea interesante es explorar el comportamiento de hacer las reservas combinando mecanismos de varios backends es algo que no se ha explorado en este trabajo, esto por ejemplo podría significar reservar memoria alineada para un dispositivo FPGA y posteriormente convertirla en pinned para CUDA usando la función apropiada para ello.

6.2.3 Profiling en profundidad

Pese a que el modelo propuesto cumple las expectativas, algunos de los resultados del estudio experimental muestran una perdida de rendimiento en ciertos casos con respecto al modelo previo. Estos resultados requieren de un análisis más detallado y profundo al realizado en este trabajo.

En particular, los retrasos introducidos en el lanzamiento de kernels y transferencias de memoria, normalmente ocultos por las esperas a las dependencias de esas operaciones, son un potencial problema que parece complicado de resolver con el modelo propuesto. Una forma de mitigar este problema podría ser usar un sistema de sincronización basado en los eventos y streams/colas del modelo nativo cuando los dispositivos que se vayan a utilizar lo permitan y solo usar la sincronización en el host cuando se vayan a utilizar dispositivos incompatibles.

6.2.4 Modelo alternativo de sincronización

En la sección 3.2.2 se menciona la opción de realizar las sincronizaciones en CUDA con las *stream memory operations*. Como se menciona, esta opción se descartó en este trabajo, ya que en el momento de la implementación estas operaciones estaban desactivadas por defecto en el driver. Desde agosto de 2022, según la documentación de CUDA, esto ya no es así. Por lo tanto, esta opción podría ser una alternativa viable y quizá solvente alguno de los problemas que tiene el modelo actual y merezca la pena hacer una prueba tentativa para ver su comportamiento.

Apéndices

Hotspot (manticore, CPU)					
size	versión	media	mediana	desviación	IC 95%
1000	Ctrl multi Async No copy	0.2524	0.2494	0.0146	[0.2470, 0.2578]
	Ctrl multi Async Copy	0.2791	0.2767	0.0165	[0.2728, 0.2853]
	Ctrl base Async No copy	0.2503	0.2573	0.0228	[0.2420, 0.2586]
	Ctrl base Async Copy	0.2830	0.2786	0.0222	[0.2749, 0.2910]
	Ctrl base Sync No copy	0.3104	0.3113	0.0177	[0.3040, 0.3169]
	Ctrl base Sync Copy	0.3821	0.3813	0.0175	[0.3757, 0.3885]
	Ctrl multi Sync No copy	0.3111	0.3104	0.0193	[0.3040, 0.3181]
	Ctrl multi Sync Copy	0.3835	0.3813	0.0129	[0.3787, 0.3883]
	Ref Sync	0.2260	0.2226	0.0251	[0.2169, 0.2352]
2500	Ctrl multi Async No copy	0.8971	0.8739	0.0772	[0.8690, 0.9252]
	Ctrl multi Async Copy	1.3982	1.3260	0.2006	[1.3225, 1.4739]
	Ctrl base Async No copy	1.1214	0.9986	0.2986	[1.0127, 1.2301]
	Ctrl base Async Copy	1.2929	1.2809	0.0501	[1.2746, 1.3111]
	Ctrl base Sync No copy	1.5504	1.4402	0.2962	[1.4426, 1.6582]
	Ctrl base Sync Copy	1.9679	1.9685	0.0573	[1.9471, 1.9888]
	Ctrl multi Sync No copy	1.3810	1.3580	0.0785	[1.3524, 1.4095]
	Ctrl multi Sync Copy	2.0854	2.0193	0.2248	[1.9973, 2.1735]
	Ref Sync	1.0969	1.0954	0.0456	[1.0803, 1.1135]

Continua en la siguiente página

					Continuación de la anterior página
5000	Ctrl multi Async No copy	3.6720	3.6521	0.1453	[3.6191, 3.7248]
	Ctrl multi Async Copy	6.6013	6.6173	0.0984	[6.5635, 6.6392]
	Ctrl base Async No copy	3.7221	3.7251	0.1365	[3.6724, 3.7718]
	Ctrl base Async Copy	6.6628	6.6502	0.0782	[6.6338, 6.6917]
	Ctrl base Sync No copy	5.5383	5.5467	0.2405	[5.4508, 5.6258]
	Ctrl base Sync Copy	8.8860	8.8710	0.1214	[8.8419, 8.9302]
	Ctrl multi Sync No copy	5.5440	5.5049	0.3144	[5.4296, 5.6585]
	Ctrl multi Sync Copy	8.8870	8.8805	0.1421	[8.8334, 8.9406]
	Ref Sync	5.8885	5.8424	0.1264	[5.8425, 5.9345]
7500	Ctrl multi Async No copy	8.5848	8.6125	0.3916	[8.4423, 8.7273]
	Ctrl multi Async Copy	16.4636	16.5182	0.4073	[16.3153, 16.6118]
	Ctrl base Async No copy	8.6243	8.6109	0.3391	[8.5009, 8.7477]
	Ctrl base Async Copy	16.5981	16.6352	0.2877	[16.4934, 16.7028]
	Ctrl base Sync No copy	12.1414	12.2501	0.4436	[11.9799, 12.3029]
	Ctrl base Sync Copy	21.8984	21.9288	0.2377	[21.8119, 21.9849]
	Ctrl multi Sync No copy	11.7115	11.5664	0.4983	[11.5302, 11.8929]
	Ctrl multi Sync Copy	21.8126	21.8372	0.4202	[21.6570, 21.9683]
	Ref Sync	13.0980	13.1109	0.1542	[13.0419, 13.1541]
10000	Ctrl multi Async No copy	14.7242	14.7821	0.5329	[14.5303, 14.9182]
	Ctrl multi Async Copy	29.1473	29.2006	0.6926	[28.8952, 29.3994]
	Ctrl base Async No copy	14.2343	14.2068	0.3908	[14.0921, 14.3765]
	Ctrl base Async Copy	28.6457	28.7657	0.5628	[28.4409, 28.8505]
	Ctrl base Sync No copy	20.7355	20.6811	0.4530	[20.5706, 20.9004]
	Ctrl base Sync Copy	38.0524	38.0538	0.3682	[37.9136, 38.1913]
	Ctrl multi Sync No copy	20.7731	20.7201	0.3943	[20.6271, 20.9192]
	Ctrl multi Sync Copy	37.8953	37.8270	0.4173	[37.7407, 38.0498]
	Ref Sync	23.6887	23.6972	0.1824	[23.6223, 23.7550]

Cuadro 1: Resultados de Hotspot CPU. Tiempos en segundos.

Hotspot (manticore, CUDA)					
size	versión	media	mediana	desviación	IC 95%
1000	Ctrl multi Async	0.0486	0.0486	0.0013	[0.0481, 0.0490]
	Ctrl base Async	0.0461	0.0460	0.0012	[0.0456, 0.0465]
	Ctrl base Sync	0.1039	0.1038	0.0009	[0.1036, 0.1042]
	Ctrl multi Sync	0.1020	0.1019	0.0005	[0.1018, 0.1021]
	Ref Async	0.0534	0.0534	0.0005	[0.0532, 0.0536]
	Ref Sync	0.0971	0.0972	0.0005	[0.0970, 0.0973]
2500	Ctrl multi Async	0.4347	0.4336	0.0082	[0.4316, 0.4378]
	Ctrl base Async	0.4237	0.4235	0.0081	[0.4207, 0.4267]
	Ctrl base Sync	0.7422	0.7436	0.0090	[0.7389, 0.7456]
	Ctrl multi Sync	0.7437	0.7432	0.0131	[0.7389, 0.7485]
	Ref Async	0.4324	0.4329	0.0087	[0.4292, 0.4357]
	Ref Sync	0.7285	0.7302	0.0099	[0.7249, 0.7321]
5000	Ctrl multi Async	1.9942	2.0043	0.0256	[1.9849, 2.0035]
	Ctrl base Async	1.9484	1.9568	0.0208	[1.9408, 1.9560]
	Ctrl base Sync	3.2156	3.2248	0.0255	[3.2064, 3.2249]
	Ctrl multi Sync	3.2804	3.2913	0.0353	[3.2676, 3.2932]
	Ref Async	1.9381	1.9487	0.0258	[1.9288, 1.9475]
	Ref Sync	3.1927	3.2053	0.0258	[3.1833, 3.2021]
10000	Ctrl multi Async	8.3417	8.3928	0.0924	[8.3081, 8.3754]
	Ctrl base Async	8.2353	8.2472	0.0365	[8.2210, 8.2496]
	Ctrl base Sync	13.1656	13.2167	0.1071	[13.1266, 13.2046]
	Ctrl multi Sync	13.4631	13.5017	0.1013	[13.4262, 13.5000]
	Ref Async	8.1178	8.1482	0.0889	[8.0854, 8.1501]
	Ref Sync	13.1189	13.1396	0.0563	[13.0976, 13.1401]
15000	Ctrl multi Async	18.9534	19.0211	0.2135	[18.8757, 19.0311]
	Ctrl base Async	18.7137	18.7274	0.1220	[18.6685, 18.7589]
	Ctrl base Sync	29.7734	29.8953	0.2542	[29.6809, 29.8659]
	Ctrl multi Sync	30.5251	30.5775	0.1754	[30.4589, 30.5913]
	Ref Async	18.4399	18.4906	0.2519	[18.3482, 18.5316]
	Ref Sync	29.8030	29.8026	0.0766	[29.7730, 29.8330]

Cuadro 2: Resultados de Hotspot CUDA. Tiempos en segundos.

Hotspot (manticore, OpenCL Nvidia)					
size	versión	media	mediana	desviación	IC 95%
1000	Ctrl multi Async	0.0478	0.0478	0.0007	[0.0475, 0.0481]
	Ctrl base Async	0.0444	0.0443	0.0004	[0.0442, 0.0445]
	Ctrl base Sync	0.0992	0.0991	0.0002	[0.0991, 0.0993]
	Ctrl multi Sync	0.1025	0.1025	0.0005	[0.1023, 0.1026]
	Ref Async	0.0563	0.0567	0.0013	[0.0558, 0.0567]
	Ref Sync	0.0964	0.0964	0.0003	[0.0963, 0.0965]
2500	Ctrl multi Async	0.4390	0.4394	0.0126	[0.4344, 0.4436]
	Ctrl base Async	0.4160	0.4162	0.0081	[0.4131, 0.4190]
	Ctrl base Sync	0.7255	0.7251	0.0065	[0.7230, 0.7279]
	Ctrl multi Sync	0.7536	0.7547	0.0075	[0.7507, 0.7564]
	Ref Async	0.4241	0.4238	0.0073	[0.4215, 0.4268]
	Ref Sync	0.7229	0.7219	0.0094	[0.7195, 0.7263]
5000	Ctrl multi Async	1.9979	2.0096	0.0271	[1.9881, 2.0078]
	Ctrl base Async	1.9597	1.9593	0.0056	[1.9575, 1.9620]
	Ctrl base Sync	3.1964	3.2036	0.0251	[3.1871, 3.2057]
	Ctrl multi Sync	3.2920	3.2939	0.0123	[3.2871, 3.2969]
	Ref Async	1.9607	1.9707	0.0230	[1.9523, 1.9690]
	Ref Sync	3.1798	3.1817	0.0103	[3.1757, 3.1838]
10000	Ctrl multi Async	8.4071	8.4091	0.0188	[8.3995, 8.4148]
	Ctrl base Async	8.1898	8.2014	0.0557	[8.1684, 8.2112]
	Ctrl base Sync	13.0950	13.1467	0.1060	[13.0565, 13.1336]
	Ctrl multi Sync	13.4839	13.4935	0.0388	[13.4687, 13.4991]
	Ref Async	8.1967	8.2053	0.0470	[8.1786, 8.2148]
	Ref Sync	13.0389	13.0478	0.0513	[13.0188, 13.0590]
15000	Ctrl multi Async	19.0735	19.0901	0.1114	[19.0289, 19.1181]
	Ctrl base Async	18.6879	18.7082	0.1350	[18.6360, 18.7398]
	Ctrl base Sync	29.6705	29.7237	0.1968	[29.5989, 29.7421]
	Ctrl multi Sync	30.5010	30.4980	0.0930	[30.4646, 30.5375]
	Ref Async	18.6988	18.7715	0.1905	[18.6294, 18.7681]
	Ref Sync	29.5515	29.5689	0.1464	[29.4952, 29.6077]

Cuadro 3: Resultados de Hotspot OpenCL Nvidia. Tiempos en segundos.

Hotspot (manticore, OpenCL AMD)					
size	versión	media	mediana	desviación	IC 95%
1000	Ctrl multi Async	0.0913	0.0897	0.0028	[0.0901, 0.0924]
	Ctrl base Async	0.0887	0.0870	0.0034	[0.0874, 0.0900]
	Ctrl base Sync	0.3906	0.3855	0.0136	[0.3856, 0.3956]
	Ctrl multi Sync	0.2110	0.2098	0.0060	[0.2087, 0.2133]
	Ref Async	0.0934	0.0931	0.0023	[0.0925, 0.0943]
	Ref Sync	0.3047	0.3055	0.0033	[0.3035, 0.3060]
2500	Ctrl multi Async	0.5757	0.5755	0.0061	[0.5734, 0.5781]
	Ctrl base Async	0.5645	0.5649	0.0057	[0.5623, 0.5666]
	Ctrl base Sync	1.8547	1.8545	0.0079	[1.8518, 1.8577]
	Ctrl multi Sync	1.4128	1.4165	0.0164	[1.4064, 1.4193]
	Ref Async	0.5521	0.5522	0.0037	[0.5508, 0.5535]
	Ref Sync	2.3444	2.3448	0.0108	[2.3404, 2.3484]
5000	Ctrl multi Async	2.3923	2.3949	0.0203	[2.3848, 2.3998]
	Ctrl base Async	2.4531	2.4567	0.0138	[2.4479, 2.4582]
	Ctrl base Sync	6.9811	6.9800	0.0100	[6.9771, 6.9851]
	Ctrl multi Sync	5.6182	5.6198	0.0294	[5.6075, 5.6289]
	Ref Async	2.2299	2.2300	0.0100	[2.2263, 2.2336]
	Ref Sync	7.6357	7.6354	0.0075	[7.6329, 7.6385]
10000	Ctrl multi Async	9.8718	9.8702	0.1219	[9.8275, 9.9162]
	Ctrl base Async	9.4649	9.4700	0.0382	[9.4510, 9.4788]
	Ctrl base Sync	25.1334	25.1510	0.1028	[25.0960, 25.1708]
	Ctrl multi Sync	21.0793	21.0971	0.2175	[21.0002, 21.1585]
	Ref Async	9.0445	9.0548	0.0479	[9.0271, 9.0620]
	Ref Sync	21.7021	21.7028	0.0064	[21.6998, 21.7044]
15000	Ctrl multi Async	22.7143	22.6240	0.4379	[22.5549, 22.8736]
	Ctrl base Async	21.0267	21.0282	0.1066	[20.9872, 21.0662]
	Ctrl base Sync	55.4103	55.4874	0.2211	[55.3298, 55.4907]
	Ctrl multi Sync	46.7960	46.8713	0.6031	[46.5765, 47.0155]
	Ref Async	20.4266	20.4480	0.0562	[20.4046, 20.4487]
	Ref Sync	44.1994	44.2347	0.1089	[44.1590, 44.2397]

Cuadro 4: Resultados de Hotspot OpenCL AMD. Tiempos en segundos.

Matrix Pow (manticore, CPU)					
size	versión	media	mediana	desviación	IC 95%
512	Ctrl multi Async No copy	0.2091	0.2049	0.0249	[0.2001, 0.2182]
	Ctrl multi Async Copy	0.1903	0.1892	0.0158	[0.1843, 0.1962]
	Ctrl base Async No copy	0.1889	0.1896	0.0206	[0.1814, 0.1964]
	Ctrl base Async Copy	0.1763	0.1767	0.0229	[0.1680, 0.1847]
	Ctrl base Sync No copy	0.2311	0.2325	0.0177	[0.2247, 0.2375]
	Ctrl base Sync Copy	0.2173	0.2172	0.0199	[0.2101, 0.2245]
	Ctrl multi Sync No copy	0.2428	0.2465	0.0215	[0.2350, 0.2507]
	Ctrl multi Sync Copy	0.2455	0.2414	0.0195	[0.2385, 0.2526]
	Ref Sync	0.2956	0.3004	0.0177	[0.2889, 0.3023]
1024	Ctrl multi Async No copy	1.0939	1.0933	0.0151	[1.0883, 1.0995]
	Ctrl multi Async Copy	1.0260	1.0232	0.0152	[1.0203, 1.0318]
	Ctrl base Async No copy	1.0736	1.0795	0.0202	[1.0660, 1.0812]
	Ctrl base Async Copy	0.9111	0.9126	0.0257	[0.9015, 0.9208]
	Ctrl base Sync No copy	1.2327	1.2315	0.0242	[1.2239, 1.2415]
	Ctrl base Sync Copy	1.0964	1.0992	0.0359	[1.0834, 1.1095]
	Ctrl multi Sync No copy	1.2652	1.2634	0.0116	[1.2607, 1.2697]
	Ctrl multi Sync Copy	1.2180	1.2177	0.0187	[1.2108, 1.2252]
	Ref Sync	1.3387	1.3526	0.0286	[1.3282, 1.3491]
1536	Ctrl multi Async No copy	2.3277	2.3248	0.0379	[2.3139, 2.3415]
	Ctrl multi Async Copy	2.2642	2.2654	0.0176	[2.2577, 2.2707]
	Ctrl base Async No copy	2.2000	2.1966	0.0335	[2.1879, 2.2122]
	Ctrl base Async Copy	1.9358	1.9381	0.0254	[1.9265, 1.9450]
	Ctrl base Sync No copy	2.5684	2.5679	0.0235	[2.5598, 2.5769]
	Ctrl base Sync Copy	2.3848	2.3809	0.0215	[2.3768, 2.3927]
	Ctrl multi Sync No copy	2.7099	2.7079	0.0257	[2.7005, 2.7192]
	Ctrl multi Sync Copy	2.7245	2.7195	0.0199	[2.7172, 2.7317]
	Ref Sync	2.7987	2.7949	0.0255	[2.7894, 2.8080]

Continua en la siguiente página

					Continuación de la anterior página
2048	Ctrl multi Async No copy	6.0332	6.0154	0.2054	[5.9584, 6.1079]
	Ctrl multi Async Copy	5.8633	5.8689	0.1452	[5.8105, 5.9162]
	Ctrl base Async No copy	6.0632	6.0574	0.1833	[5.9953, 6.1311]
	Ctrl base Async Copy	5.2346	5.2639	0.1027	[5.1973, 5.2720]
	Ctrl base Sync No copy	6.6453	6.6555	0.1884	[6.5767, 6.7138]
	Ctrl base Sync Copy	6.0211	5.9928	0.1215	[5.9769, 6.0653]
	Ctrl multi Sync No copy	6.7597	6.7213	0.1951	[6.6887, 6.8307]
	Ctrl multi Sync Copy	6.6085	6.5961	0.1076	[6.5679, 6.6491]
	Ref Sync	7.3153	7.3186	0.0424	[7.2999, 7.3307]
3072	Ctrl multi Async No copy	16.6351	16.6309	0.4105	[16.4857, 16.7845]
	Ctrl multi Async Copy	16.3821	16.4052	0.3687	[16.2479, 16.5163]
	Ctrl base Async No copy	16.3410	16.3402	0.4898	[16.1595, 16.5224]
	Ctrl base Async Copy	14.8612	14.8809	0.2764	[14.7606, 14.9618]
	Ctrl base Sync No copy	17.8134	17.8009	0.4222	[17.6597, 17.9671]
	Ctrl base Sync Copy	16.6139	16.6323	0.2405	[16.5264, 16.7014]
	Ctrl multi Sync No copy	18.2492	18.2297	0.2852	[18.1454, 18.3530]
	Ctrl multi Sync Copy	17.9618	17.9917	0.2395	[17.8679, 18.0557]
	Ref Sync	21.6211	21.6282	0.0416	[21.6057, 21.6365]
4096	Ctrl multi Async No copy	37.2912	37.2467	0.6417	[37.0535, 37.5289]
	Ctrl multi Async Copy	34.8667	35.0171	0.9406	[34.5243, 35.2090]
	Ctrl base Async No copy	36.7860	36.6468	0.5228	[36.5957, 36.9763]
	Ctrl base Async Copy	32.8634	32.7888	0.6518	[32.6220, 33.1049]
	Ctrl base Sync No copy	39.1980	39.2694	0.4393	[39.0322, 39.3637]
	Ctrl base Sync Copy	35.9129	36.0417	0.4992	[35.7313, 36.0946]
	Ctrl multi Sync No copy	40.0918	40.1173	0.5766	[39.8819, 40.3016]
	Ctrl multi Sync Copy	37.4217	37.1830	0.8154	[37.1249, 37.7185]
	Ref Sync	54.2232	54.2339	0.0946	[54.1876, 54.2589]

Cuadro 5: Resultados de Matrix Pow CPU. Tiempos en segundos.

Matrix Pow (manticore, CUDA)					
size	versión	media	mediana	desviación	IC 95%
1024	Ctrl multi Async	0.1517	0.1517	0.0005	[0.1515, 0.1519]
	Ctrl base Async	0.1434	0.1434	0.0004	[0.1432, 0.1435]
	Ctrl base Sync	0.1865	0.1864	0.0007	[0.1863, 0.1868]
	Ctrl multi Sync	0.1933	0.1934	0.0007	[0.1931, 0.1936]
	Ref Async	0.1637	0.1636	0.0004	[0.1635, 0.1638]
	Ref Sync	0.1851	0.1851	0.0001	[0.1850, 0.1851]
1536	Ctrl multi Async	0.3461	0.3460	0.0012	[0.3456, 0.3465]
	Ctrl base Async	0.3235	0.3231	0.0013	[0.3231, 0.3240]
	Ctrl base Sync	0.4463	0.4462	0.0003	[0.4461, 0.4464]
	Ctrl multi Sync	0.4639	0.4642	0.0015	[0.4633, 0.4646]
	Ref Async	0.3625	0.3620	0.0023	[0.3616, 0.3634]
	Ref Sync	0.4463	0.4462	0.0007	[0.4461, 0.4466]
2048	Ctrl multi Async	0.6367	0.6301	0.0124	[0.6322, 0.6412]
	Ctrl base Async	0.5866	0.5866	0.0022	[0.5857, 0.5874]
	Ctrl base Sync	0.8568	0.8562	0.0029	[0.8556, 0.8579]
	Ctrl multi Sync	0.8878	0.8883	0.0022	[0.8869, 0.8887]
	Ref Async	0.6533	0.6533	0.0029	[0.6521, 0.6545]
	Ref Sync	0.8561	0.8561	0.0011	[0.8557, 0.8565]
4096	Ctrl multi Async	2.6243	2.6229	0.0078	[2.6210, 2.6277]
	Ctrl base Async	2.4511	2.4499	0.0091	[2.4473, 2.4549]
	Ctrl base Sync	4.2239	4.2168	0.0315	[4.2123, 4.2356]
	Ctrl multi Sync	4.3302	4.3309	0.0141	[4.3244, 4.3359]
	Ref Async	2.6805	2.6795	0.0058	[2.6778, 2.6831]
	Ref Sync	4.2158	4.2143	0.0065	[4.2134, 4.2183]
6144	Ctrl multi Async	6.5390	6.5385	0.0024	[6.5381, 6.5399]
	Ctrl base Async	6.5438	6.5440	0.0010	[6.5434, 6.5442]
	Ctrl base Sync	12.4080	12.4064	0.0647	[12.3844, 12.4315]
	Ctrl multi Sync	12.5076	12.5178	0.0347	[12.4934, 12.5218]
	Ref Async	6.7627	6.7612	0.0068	[6.7600, 6.7654]
	Ref Sync	12.2656	12.2624	0.0108	[12.2613, 12.2699]
8192	Ctrl multi Async	15.4849	15.4850	0.0039	[15.4835, 15.4864]
	Ctrl base Async	15.4852	15.4859	0.0034	[15.4840, 15.4865]
	Ctrl base Sync	25.9939	26.0274	0.1098	[25.9539, 26.0338]
	Ctrl multi Sync	26.1621	26.1643	0.0767	[26.1314, 26.1928]
	Ref Async	15.5912	15.5905	0.0087	[15.5877, 15.5947]
	Ref Sync	25.7003	25.6987	0.0186	[25.6929, 25.7077]

Cuadro 6: Resultados de Matrix Pow CUDA. Tiempos en segundos.

Matrix Pow (manticore, OpenCL Nvidia)					
size	versión	media	mediana	desviación	IC 95%
1024	Ctrl multi Async	0.1463	0.1463	0.0002	[0.1462, 0.1464]
	Ctrl base Async	0.1452	0.1451	0.0003	[0.1451, 0.1453]
	Ctrl base Sync	0.1946	0.1944	0.0006	[0.1944, 0.1948]
	Ctrl multi Sync	0.1959	0.1958	0.0003	[0.1958, 0.1960]
	Ref Async	0.1622	0.1626	0.0033	[0.1610, 0.1634]
	Ref Sync	0.1944	0.1944	0.0002	[0.1944, 0.1945]
1536	Ctrl multi Async	0.3352	0.3352	0.0022	[0.3344, 0.3360]
	Ctrl base Async	0.3273	0.3272	0.0011	[0.3269, 0.3278]
	Ctrl base Sync	0.4741	0.4739	0.0007	[0.4738, 0.4743]
	Ctrl multi Sync	0.4801	0.4798	0.0011	[0.4797, 0.4805]
	Ref Async	0.3547	0.3557	0.0042	[0.3531, 0.3562]
	Ref Sync	0.4753	0.4751	0.0008	[0.4750, 0.4756]
2048	Ctrl multi Async	0.6102	0.6102	0.0032	[0.6090, 0.6115]
	Ctrl base Async	0.5951	0.5953	0.0020	[0.5943, 0.5960]
	Ctrl base Sync	0.9212	0.9214	0.0013	[0.9207, 0.9217]
	Ctrl multi Sync	0.9371	0.9374	0.0019	[0.9364, 0.9379]
	Ref Async	0.6323	0.6334	0.0081	[0.6293, 0.6352]
	Ref Sync	0.9265	0.9263	0.0022	[0.9257, 0.9273]
4096	Ctrl multi Async	2.5579	2.5574	0.0075	[2.5548, 2.5611]
	Ctrl base Async	2.5097	2.4843	0.0563	[2.4892, 2.5302]
	Ctrl base Sync	4.6644	4.6651	0.0230	[4.6550, 4.6738]
	Ctrl multi Sync	4.7407	4.7426	0.0093	[4.7371, 4.7444]
	Ref Async	2.5441	2.5304	0.0380	[2.5303, 2.5580]
	Ref Sync	4.6978	4.6947	0.0119	[4.6932, 4.7024]
6144	Ctrl multi Async	7.2118	7.2038	0.0155	[7.2062, 7.2175]
	Ctrl base Async	7.2374	7.2479	0.0253	[7.2282, 7.2466]
	Ctrl base Sync	12.9863	12.9667	0.0597	[12.9629, 13.0097]
	Ctrl multi Sync	13.1332	13.1257	0.0335	[13.1206, 13.1459]
	Ref Async	7.2384	7.2393	0.0056	[7.2363, 7.2405]
	Ref Sync	12.9995	12.9948	0.0254	[12.9898, 13.0093]
8192	Ctrl multi Async	17.2116	17.2170	0.0229	[17.2033, 17.2200]
	Ctrl base Async	17.2563	17.2666	0.0417	[17.2400, 17.2726]
	Ctrl base Sync	27.5781	27.5198	0.1942	[27.5074, 27.6488]
	Ctrl multi Sync	27.7419	27.7435	0.0563	[27.7198, 27.7639]
	Ref Async	17.2701	17.2670	0.0199	[17.2626, 17.2776]
	Ref Sync	27.4624	27.4629	0.0487	[27.4440, 27.4807]

Cuadro 7: Resultados de Matrix Pow OpenCL Nvidia. Tiempos en segundos.

Matrix Pow (manticore, OpenCL AMD)					
size	versión	media	mediana	desviación	IC 95%
1024	Ctrl multi Async	0.1653	0.1651	0.0012	[0.1649, 0.1657]
	Ctrl base Async	0.1603	0.1601	0.0015	[0.1597, 0.1609]
	Ctrl base Sync	0.4138	0.4119	0.0201	[0.4065, 0.4211]
	Ctrl multi Sync	0.2664	0.2645	0.0057	[0.2643, 0.2685]
	Ref Async	0.1757	0.1760	0.0016	[0.1751, 0.1763]
	Ref Sync	0.4093	0.4108	0.0122	[0.4048, 0.4137]
1536	Ctrl multi Async	0.3690	0.3686	0.0027	[0.3680, 0.3700]
	Ctrl base Async	0.3562	0.3566	0.0062	[0.3540, 0.3585]
	Ctrl base Sync	1.0273	1.0228	0.0333	[1.0152, 1.0394]
	Ctrl multi Sync	0.6361	0.6356	0.0035	[0.6348, 0.6373]
	Ref Async	0.3622	0.3614	0.0022	[0.3613, 0.3630]
	Ref Sync	1.1251	1.1366	0.0458	[1.1084, 1.1418]
2048	Ctrl multi Async	0.6679	0.6682	0.0045	[0.6663, 0.6695]
	Ctrl base Async	0.6479	0.6469	0.0066	[0.6455, 0.6504]
	Ctrl base Sync	2.0255	2.0253	0.0041	[2.0239, 2.0270]
	Ctrl multi Sync	1.2643	1.2639	0.0066	[1.2619, 1.2667]
	Ref Async	0.6372	0.6375	0.0030	[0.6361, 0.6383]
	Ref Sync	2.0282	2.0280	0.0036	[2.0269, 2.0295]
4096	Ctrl multi Async	3.5188	3.5137	0.0241	[3.5100, 3.5275]
	Ctrl base Async	3.5624	3.5563	0.0299	[3.5511, 3.5737]
	Ctrl base Sync	7.3003	7.2991	0.0058	[7.2982, 7.3025]
	Ctrl multi Sync	6.6496	6.6448	0.0226	[6.6412, 6.6579]
	Ref Async	4.0639	4.0638	0.0018	[4.0632, 4.0645]
	Ref Sync	7.6764	7.6863	0.0259	[7.6669, 7.6858]
6144	Ctrl multi Async	11.9748	11.9808	0.0434	[11.9590, 11.9906]
	Ctrl base Async	12.0516	12.0251	0.0557	[12.0306, 12.0726]
	Ctrl base Sync	19.1439	19.1238	0.0819	[19.1141, 19.1737]
	Ctrl multi Sync	18.7283	18.7192	0.0804	[18.6985, 18.7580]
	Ref Async	13.4702	13.4705	0.0031	[13.4691, 13.4713]
	Ref Sync	20.2284	20.2451	0.0343	[20.2159, 20.2408]
8192	Ctrl multi Async	27.7584	27.7546	0.0340	[27.7460, 27.7707]
	Ctrl base Async	27.8016	27.7807	0.0522	[27.7820, 27.8213]
	Ctrl base Sync	39.7619	39.7351	0.1310	[39.7142, 39.8095]
	Ctrl multi Sync	39.6742	39.6820	0.0887	[39.6414, 39.7071]
	Ref Async	31.9335	31.9285	0.0147	[31.9279, 31.9390]
	Ref Sync	43.0950	43.1132	0.0649	[43.0713, 43.1186]

Cuadro 8: Resultados de Matrix Pow OpenCL AMD. Tiempos en segundos.

Sobel YUV (manticore, CPU)					
size	versión	media	mediana	desviación	IC 95%
FTF	Ctrl multi Async No copy	0.5218	0.5231	0.0223	[0.5137, 0.5299]
	Ctrl multi Async Copy	0.2883	0.2952	0.0262	[0.2788, 0.2979]
	Ctrl base Async No copy	0.4906	0.4877	0.0202	[0.4833, 0.4980]
	Ctrl base Async Copy	0.2872	0.2882	0.0202	[0.2798, 0.2947]
	Ctrl base Sync No copy	0.4908	0.4921	0.0164	[0.4846, 0.4970]
	Ctrl base Sync Copy	0.5439	0.5425	0.0149	[0.5385, 0.5494]
	Ctrl multi Sync No copy	0.5187	0.5203	0.0168	[0.5126, 0.5248]
	Ctrl multi Sync Copy	0.5628	0.5588	0.0154	[0.5572, 0.5684]
	Ref Sync	0.4363	0.4360	0.0119	[0.4318, 0.4408]
MTF	Ctrl multi Async No copy	0.5017	0.5028	0.0173	[0.4954, 0.5080]
	Ctrl multi Async Copy	0.2823	0.2820	0.0230	[0.2740, 0.2907]
	Ctrl base Async No copy	0.4949	0.4926	0.0151	[0.4893, 0.5005]
	Ctrl base Async Copy	0.2753	0.2806	0.0192	[0.2682, 0.2824]
	Ctrl base Sync No copy	0.4963	0.4919	0.0191	[0.4893, 0.5032]
	Ctrl base Sync Copy	0.5639	0.5641	0.0177	[0.5575, 0.5704]
	Ctrl multi Sync No copy	0.5040	0.5011	0.0200	[0.4967, 0.5113]
	Ctrl multi Sync Copy	0.5577	0.5589	0.0122	[0.5532, 0.5621]
	Ref Sync	0.4322	0.4364	0.0171	[0.4259, 0.4386]
MTM	Ctrl multi Async No copy	0.5426	0.5415	0.0172	[0.5363, 0.5490]
	Ctrl multi Async Copy	0.3230	0.3253	0.0182	[0.3164, 0.3296]
	Ctrl base Async No copy	0.5391	0.5391	0.0134	[0.5342, 0.5441]
	Ctrl base Async Copy	0.3183	0.3170	0.0181	[0.3117, 0.3249]
	Ctrl base Sync No copy	0.5418	0.5448	0.0144	[0.5366, 0.5471]
	Ctrl base Sync Copy	0.6205	0.6177	0.0200	[0.6132, 0.6278]
	Ctrl multi Sync No copy	0.5413	0.5420	0.0115	[0.5368, 0.5457]
	Ctrl multi Sync Copy	0.6174	0.6176	0.0162	[0.6114, 0.6234]
	Ref Sync	0.5078	0.5102	0.0221	[0.4997, 0.5158]
FTM	Ctrl multi Async No copy	0.5610	0.5603	0.0152	[0.5555, 0.5665]
	Ctrl multi Async Copy	0.3226	0.3207	0.0132	[0.3177, 0.3275]
	Ctrl base Async No copy	0.5307	0.5297	0.0106	[0.5268, 0.5347]
	Ctrl base Async Copy	0.3164	0.3135	0.0156	[0.3106, 0.3222]
	Ctrl base Sync No copy	0.5424	0.5428	0.0147	[0.5370, 0.5477]
	Ctrl base Sync Copy	0.6060	0.6080	0.0121	[0.6016, 0.6104]
	Ctrl multi Sync No copy	0.5559	0.5541	0.0118	[0.5516, 0.5602]
	Ctrl multi Sync Copy	0.6321	0.6313	0.0131	[0.6272, 0.6369]
	Ref Sync	0.5094	0.5071	0.0163	[0.5035, 0.5153]

Cuadro 9: Resultados de Sobel YUV CPU. Tiempos en segundos.

Sobel YUV (manticore, CUDA)					
size	versión	media	mediana	desviación	IC 95%
FTF	Ctrl multi Async	0.2112	0.2112	0.0016	[0.2106, 0.2118]
	Ctrl base Async	0.1831	0.1833	0.0011	[0.1827, 0.1835]
	Ctrl base Sync	0.2420	0.2418	0.0009	[0.2416, 0.2423]
	Ctrl multi Sync	0.2688	0.2688	0.0010	[0.2685, 0.2692]
	Ref Async	0.1886	0.1887	0.0010	[0.1882, 0.1889]
	Ref Sync	0.2269	0.2268	0.0004	[0.2267, 0.2271]
MTF	Ctrl multi Async	0.1937	0.1939	0.0008	[0.1934, 0.1940]
	Ctrl base Async	0.1698	0.1697	0.0006	[0.1696, 0.1701]
	Ctrl base Sync	0.2310	0.2312	0.0007	[0.2307, 0.2313]
	Ctrl multi Sync	0.2505	0.2505	0.0004	[0.2504, 0.2507]
	Ref Async	0.1818	0.1819	0.0015	[0.1812, 0.1823]
	Ref Sync	0.2203	0.2198	0.0016	[0.2197, 0.2208]
MTM	Ctrl multi Async	0.1231	0.1230	0.0007	[0.1228, 0.1234]
	Ctrl base Async	0.1116	0.1118	0.0008	[0.1113, 0.1119]
	Ctrl base Sync	0.1743	0.1743	0.0003	[0.1742, 0.1744]
	Ctrl multi Sync	0.1763	0.1763	0.0006	[0.1761, 0.1765]
	Ref Async	0.1196	0.1197	0.0008	[0.1193, 0.1199]
	Ref Sync	0.1651	0.1651	0.0003	[0.1650, 0.1652]
FTM	Ctrl multi Async	0.1359	0.1358	0.0011	[0.1355, 0.1363]
	Ctrl base Async	0.1237	0.1238	0.0008	[0.1234, 0.1240]
	Ctrl base Sync	0.1860	0.1861	0.0007	[0.1858, 0.1863]
	Ctrl multi Sync	0.1904	0.1904	0.0009	[0.1901, 0.1908]
	Ref Async	0.1266	0.1268	0.0007	[0.1264, 0.1269]
	Ref Sync	0.1754	0.1755	0.0004	[0.1753, 0.1756]

Cuadro 10: Resultados de Sobel YUV CUDA. Tiempos en segundos.

Sobel YUV (manticore, OpenCL Nvidia)					
size	versión	media	mediana	desviación	IC 95%
FTF	Ctrl multi Async	0.2105	0.2105	0.0016	[0.2099, 0.2111]
	Ctrl base Async	0.1810	0.1813	0.0012	[0.1806, 0.1815]
	Ctrl base Sync	0.2310	0.2309	0.0011	[0.2306, 0.2314]
	Ctrl multi Sync	0.2678	0.2678	0.0013	[0.2673, 0.2683]
	Ref Async	0.1854	0.1852	0.0016	[0.1849, 0.1860]
	Ref Sync	0.2277	0.2278	0.0004	[0.2275, 0.2279]
MTF	Ctrl multi Async	0.2003	0.2004	0.0009	[0.2000, 0.2006]
	Ctrl base Async	0.1702	0.1701	0.0011	[0.1698, 0.1706]
	Ctrl base Sync	0.2198	0.2199	0.0005	[0.2196, 0.2200]
	Ctrl multi Sync	0.2565	0.2566	0.0005	[0.2563, 0.2567]
	Ref Async	0.1817	0.1833	0.0043	[0.1801, 0.1833]
	Ref Sync	0.2249	0.2250	0.0006	[0.2247, 0.2252]
MTM	Ctrl multi Async	0.2749	0.2750	0.0011	[0.2745, 0.2753]
	Ctrl base Async	0.2444	0.2446	0.0007	[0.2442, 0.2447]
	Ctrl base Sync	0.2940	0.2939	0.0013	[0.2935, 0.2945]
	Ctrl multi Sync	0.3308	0.3306	0.0009	[0.3304, 0.3311]
	Ref Async	0.2580	0.2584	0.0038	[0.2566, 0.2594]
	Ref Sync	0.2976	0.2977	0.0008	[0.2974, 0.2979]
FTM	Ctrl multi Async	0.2852	0.2854	0.0008	[0.2849, 0.2855]
	Ctrl base Async	0.2560	0.2561	0.0014	[0.2555, 0.2566]
	Ctrl base Sync	0.3060	0.3062	0.0014	[0.3055, 0.3066]
	Ctrl multi Sync	0.3415	0.3418	0.0014	[0.3410, 0.3421]
	Ref Async	0.2614	0.2607	0.0030	[0.2603, 0.2625]
	Ref Sync	0.3004	0.3004	0.0004	[0.3002, 0.3005]

Cuadro 11: Resultados de Sobel YUV OpenCL Nvidia. Tiempos en segundos.

Sobel YUV (manticore, OpenCL AMD)					
size	versión	media	mediana	desviación	IC 95%
FTF	Ctrl multi Async	0.2205	0.2191	0.0126	[0.2159, 0.2251]
	Ctrl base Async	0.1669	0.1668	0.0010	[0.1666, 0.1673]
	Ctrl base Sync	0.2350	0.2356	0.0025	[0.2341, 0.2359]
	Ctrl multi Sync	0.2936	0.2932	0.0139	[0.2886, 0.2987]
	Ref Async	0.1780	0.1783	0.0020	[0.1773, 0.1788]
	Ref Sync	0.2628	0.2630	0.0029	[0.2617, 0.2639]
MTF	Ctrl multi Async	0.2040	0.2039	0.0027	[0.2030, 0.2050]
	Ctrl base Async	0.1863	0.1836	0.0068	[0.1837, 0.1890]
	Ctrl base Sync	0.2542	0.2499	0.0099	[0.2504, 0.2580]
	Ctrl multi Sync	0.2697	0.2695	0.0026	[0.2687, 0.2707]
	Ref Async	0.1972	0.1978	0.0021	[0.1964, 0.1979]
	Ref Sync	0.2900	0.2903	0.0016	[0.2893, 0.2906]
MTM	Ctrl multi Async	0.2783	0.2785	0.0017	[0.2777, 0.2789]
	Ctrl base Async	0.2598	0.2598	0.0009	[0.2595, 0.2601]
	Ctrl base Sync	0.3264	0.3271	0.0023	[0.3255, 0.3274]
	Ctrl multi Sync	0.3445	0.3445	0.0013	[0.3440, 0.3450]
	Ref Async	0.2737	0.2735	0.0013	[0.2732, 0.2742]
	Ref Sync	0.3630	0.3632	0.0031	[0.3617, 0.3642]
FTM	Ctrl multi Async	0.2970	0.2969	0.0139	[0.2920, 0.3021]
	Ctrl base Async	0.2399	0.2398	0.0007	[0.2396, 0.2402]
	Ctrl base Sync	0.3065	0.3069	0.0018	[0.3058, 0.3072]
	Ctrl multi Sync	0.3655	0.3685	0.0127	[0.3608, 0.3701]
	Ref Async	0.2517	0.2514	0.0023	[0.2508, 0.2525]
	Ref Sync	0.3299	0.3297	0.0015	[0.3293, 0.3305]

Cuadro 12: Resultados de Sobel YUV OpenCL AMD. Tiempos en segundos.

Chain Matrix Multiplication (manticore)					
size	versi¾n	media	mediana	desviaci¾n	IC 95%
1024	Ctrl multi Async	0.0763	0.0745	0.0062	[0.0740, 0.0786]
	CUDA Ref Async	0.0638	0.0634	0.0017	[0.0632, 0.0644]
	OpenCL AMD Ref Async	0.1043	0.1043	0.0008	[0.1041, 0.1046]
	OpenCL Ref Async	0.0765	0.0764	0.0001	[0.0764, 0.0765]
1536	Ctrl multi Async	0.1608	0.1610	0.0123	[0.1564, 0.1653]
	CUDA Ref Async	0.2028	0.2029	0.0003	[0.2027, 0.2030]
	OpenCL AMD Ref Async	0.3358	0.3360	0.0008	[0.3355, 0.3361]
	OpenCL Ref Async	0.2455	0.2452	0.0013	[0.2451, 0.2460]
2048	Ctrl multi Async	0.3003	0.2997	0.0073	[0.2974, 0.3031]
	CUDA Ref Async	0.4426	0.4403	0.0075	[0.4399, 0.4454]
	OpenCL AMD Ref Async	0.7921	0.7918	0.0016	[0.7915, 0.7927]
	OpenCL Ref Async	0.5488	0.5489	0.0013	[0.5483, 0.5493]
4096	Ctrl multi Async	2.0395	2.0377	0.0075	[2.0368, 2.0423]
	CUDA Ref Async	3.2548	3.2547	0.0022	[3.2540, 3.2556]
	OpenCL AMD Ref Async	6.8849	6.8836	0.0111	[6.8808, 6.8890]
	OpenCL Ref Async	4.1510	4.1520	0.0028	[4.1499, 4.1521]
6144	Ctrl multi Async	6.8278	6.8224	0.0233	[6.8191, 6.8364]
	CUDA Ref Async	12.8221	12.8247	0.0067	[12.8196, 12.8245]
	OpenCL AMD Ref Async	23.2273	23.2269	0.0117	[23.2226, 23.2320]
	OpenCL Ref Async	14.1220	14.1179	0.0139	[14.1168, 14.1272]
8192	Ctrl multi Async	15.7141	15.7145	0.0261	[15.7046, 15.7236]
	CUDA Ref Async	30.5151	30.5161	0.0039	[30.5136, 30.5165]
	OpenCL AMD Ref Async	54.2581	54.2852	0.1150	[54.2162, 54.3000]
	OpenCL Ref Async	35.1118	35.0265	0.3106	[34.9875, 35.2360]

Cuadro 13: Resultados de Chain Matrix Multiplication. Tiempos en segundos.

Bibliografía

[1] Y. Torres, F. J. Andújar, A. Gonzalez-Escribano, and D. R. Llanos, "Supporting efficient overlapping of host-device operations for heterogeneous programming with ctrl-events," *Journal of Parallel and Distributed Computing*, Pendiente de publicación.

[2] "Top500 supercomputers," ultimo acceso Diciembre 2022. [Online]. Available: https://www.top500.org

[3] B. Caulfield, "What's the difference between a cpu and a gpu?" ultimo acceso diciembre 2022. [Online]. Available: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

[4] P. Sundararajan, "High performance computing using fpgas," Citeseer, Tech. Rep., 2010.

[5] A. Moreton-Fernandez, H. Ortega-Arranz, and A. Gonzalez-Escribano, "Controllers: An abstraction to ease the use of hardware accelerators." *The International Journal of High Performance Computing Applications (IJHPCA)*, vol. 32, no. 6, p. 838–853, 2018.

[6] "Página web del grupo de investigación Trasgo de la Universidad de Valladolid," ultimo acceso Diciembre de 2022. [Online]. Available: https://trasgo.infor.uva.es/

[7] NVIDIA, "CUDA Toolkit Documentation," ultimo acceso Diciembre de 2022. [Online]. Available: https://docs.nvidia.com/cuda/

[8] Intel Corporation, "Intel oneAPI Level Zero specification," ultimo acceso enero 2023. [Online]. Available: https://spec.oneapi.io/level-zero/latest/index.html

[9] AMD, "HIP Programming guide," ultimo acceso enero 2023. [Online]. Available: https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html

[10] T. K. G. Inc., "Open Computing Language (OpenCL)," ultimo acceso Diciembre de 2022. [Online]. Available: https://www.khronos.org/opencl/

[11] B. B. S. Center), "OmpSs2 Programming model," ultimo acceso Diciembre de 2022. [Online]. Available: https://pm.bsc.es/ompss-2

[12] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across many-core architectures," *2013 Extreme Scaling Workshop (xsw 2013)*, pp. 18–24, 2013.

[13] S. N. Laboratory, "Kokkos C++ Performance Portability Programming EcoSystem: The Programming Model Parallel Execution and Memory," ultimo acceso Diciembre de 2022. [Online]. Available: https://github.com/Kokkos/kokkos

[14] The Kokkos Team, "The Kokkos Lectures, Module 2: Views and Spaces," ultimo acceso Diciembre de 2022. [Online]. Available: https://github.com/kokkos/kokkos-tutorials/blob/main/LectureSeries/KokkosTutorial_02_SIMDStreamsTasking.pdf

[15] ——, "The Kokkos Lectures, Module 5: Stream, Tasking and SIMD," ultimo acceso Diciembre de 2022. [Online]. Available: https://github.com/kokkos/kokkos-tutorials/blob/main/LectureSeries/KokkosTutorial_05_SIMDStreamsTasking.pdf

[16] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2676870.2676883

[17] P. Diehl, M. Seshadri, T. Heller, and H. Kaiser, "Integration of cuda processing within the c++ library for parallelism and concurrency (hpx)," *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pp. 19–28, 2018. [Online]. Available: https://arxiv.org/abs/1810.11482

[18] T. Gysi, J. Bär, and T. Hoefler, "dCUDA: Hardware Supported Overlap of Computation and Communication," in *Proc. SC16, IEEE*, Salt Lake City, Utah, USA, 2016, pp. 609–620.

[19] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-gpu programming model for irregular computations," in *Proc. PPoPP '17, ACM*, Austin, Texas, USA, 2017, pp. 235–248.

[20] L. Wang, W. Wu, Z. Xu, J. Xiao, and Y. Yang, "Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing," in *Proc. ICS '16, ACM*, Istanbul, Turkey, 2016, pp. 20:1–20:11.

[21] R. Vasudevan, S. Vadhiyar, and L. Kalé, "G-charm: An adaptive runtime system for message-driven parallel applications on hybrid systems," in *Proc. ICS 2013, ACM*, Eugene, Oregon, USA, 2013, pp. 349–358.

[22] C++ Standards Committee Papers, "A unified executors proposal for c++," 2020, (Ultimo acceso Enero 2023). [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0443r14.html

[23] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.

[24] Lawrence Livermore National Laboratory, "RAJA Performance Portability Layer (C++)," 2020, ultimo acceso Enero 2023. [Online]. Available: https://github.com/LLNL/RAJA

[25] A. Vilches, A. Navarro, F. Corbera, A. Rodriguez, and R. Asenjo, "Heterogeneous parallel for template based on TBBs," in *Proc. HLPP'17*. Valladolid, Spain: Springer, 2017.

[26] B. Pérez, J. Bosque, and R. Beivide, "Simplifying programming and load balancing of data parallel applications on heterogeneous systems," in *Proc. GPGPU '16*. Barcelona, Spain: ACM, 2016, pp. 42–51.

[27] S. Venkatasubramanian and R. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems," in *Proc. ICS'09*. Yorktown Heights, NY, USA: ACM, 2009, pp. 244–255.

[28] P. Thoman *et al.*, "A taxonomy of task-based parallel programming technologies for high-performance computing," *The Journal of Supercomputing*, vol. 74, pp. 1422–1434, 2018.

[29] Khronos OpenCL working group, "SYCL 1.2.1 specification standard," 2020, ultimo acceso enero 2023. [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

[30] A. Murray and E. Crawford, "Compute aorta: A toolkit for implementing heterogeneous programming models," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, 2020.

[31] "Intel oneapi webpage," 2022, ultimo acceso Enero 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html

[32] "The trisycl project," 2021, (accessed November 15, 2021). [Online]. Available: https://github.com/triSYCL/triSYCL

[33] A. Alpay and V. Heuveline, "SYCL beyond OpenCL: The architecture, current state and future direction of HipSYCL," in *Proceedings of the International Workshop on OpenCL*. New York, NY, USA: Association for Computing Machinery, 2020.

[34] A. Rasch, J. Bigge, M. Wrodarczyk, R. Schulze, and S. Gorlatch, "docal: high-level distributed programming with opencl and cuda," *The Journal of Supercomputing*, vol. 76, pp. 5117–5138, 2020.

[35] G. Rodriguez-Canal, Y. Torres, F. J. Andújar, and A. Gonzalez-Escribano, "Efficient heterogeneous programming with fpgas using the controller model," *J. Supercomput.*, vol. 77, no. 12, p. 13995–14010, dec 2021. [Online]. Available: https://doi.org/10.1007/s11227-021-03792-7

[36] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D. R. Llanos, "An extensible system for multilevel automatic data partition and mapping." *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, p. 1145–1154, May 2014.

[37] "CUDA C Programming Guide: Multi-Device system, stream and event behaviour," 2021, ultimo acceso Enero 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#stream-and-event-behavior

[38] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0320-y

[39] NVIDIA Corporation, "CUDA Driver API: Stream Memory Operations," 2021, ultimo acceso Enero 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__MEMOP.html

[40] ——, "CUDA Runtime vs Driver API," ultimo acceso enero 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/driver-vs-runtime-api.html

[41] M. Harris, "How to optimize data transfers in cuda c/c++," 2012, ultimo acceso Enero 2023. [Online]. Available: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

[42] NVIDIA Corporation, "NVIDIA OpenCL Best Practices," 2022, ultimo acceso Enero 2023. [Online]. Available: https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[43] AMD Corporation, "AMD OpenCL Programming Optimization Guide," 2013, ultimo acceso Enero 2023. [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide.pdf

[44] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. USA: IEEE Computer Society, 2009, p. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[45] L.-N. Pouchet and et al., "Polybench/c, the polyhedral benchmark suite, gpu 1.0," 2012, ultimo acceso Enero de 2023. [Online]. Available: http://web.cs.ucla.edu/~pouchet/software/polybench

[46] NVIDIA, "CUDA Samples," 2022, ultimo acceso Enero 2023. [Online]. Available: https://github.com/nvidia/cuda-samples

[47] R. Gonzalez and R. Woods, *Digital Image Processing*, 3rd ed. Prentice Hall, 2007.