Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN INGENIERÍA DE SOFTWARE

Sistema de ejecución y coordinación de CPUs en el modelo Controller de programación paralela heterogénea

---

Alumno:
D. Sergio Alonso Pascual

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

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. Esta tendencia se puede ver claramente en los supercomputadores más potentes del mundo en el TOP500. Uno de los desafíos que tiene gestionar y obtener eficiencia de una aplicación ejecutada en 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 Controllers 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 Controllers soporta el uso de los modelos de programación paralelos CUDA y OpenCL con dispositivos GPU (Graphic Processing Unit).

Este trabajo propone una extensión del modelo de programación Controllers a través de un nuevo backend que permita que uno o varios núcleos de uno o varios procesadores, se utilicen como unidades de cómputo de forma transparente al usuario. También se realiza un estudio experimental para medir el rendimiento del nuevo backend.

Abstract

Currently, parallel programming bases its solutions on all types of existing hardware. This results in heterogeneous platforms that make use of coprocessors. This trend can be clearly seen in the world's most powerful supercomputers in the TOP500. 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 Controllers 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 CUDA and OpenCL with GPU (Graphic Processing Unit) devices.

This work proposes an extension of the Controllers programming model in the form of a new backend that allows one or more cores of one or more processors to be used as computing units in a transparent way to the user. An experimental study is also done to measure the performance of the new backend.

Índice general

Índice de figuras

Índice de cuadros

Listings

Capítulo 1

Introducción

1.1. Contexto

La computación heterogénea es un sistema de cómputo, el cual, está compuesto por unidades computaciones de naturaleza diferente. La computación heterogénea se utiliza para aprovechar, en la medida de los posible, 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. Para ello, la computación heterogénea integra aceleradores con distintas arquitecturas capaces de explotar las características de los problemas desde distintos enfoques obteniendo, de este modo, un mayor rendimiento.

En la lista TOP500 [3] 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) [8] o FPGAS (*Field Programable Gate Arrays*) [13]. 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 maquina 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 las operaciones que lanza el host se ejecuten de forma simultanea. El mayor inconveniente de estas llamadas es que obligan al programador a usar complejas técnicas de bajo nivel, en muchas ocasiones especificas del modelo de programación utilizado, limitando la portabilidad de los programas creados a otro tipo de aceleradores. Además de complicar aun 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.

Usar los coprocesadores es una tarea costosa ya que requiere un alto conocimiento de los detalles de los diferentes recursos hardware del dispositivo. 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 inherente de éstas sin necesidad de gestionar o seleccionar parámetros específicos de la arquitectura.

1.2. Motivación

Controllers [2, 4] es un modelo de programación paralela desarrollado por el grupo Trasgo [12] 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 si.

Actualmente Controllers soporta varias plataformas y tipos de aceleradores. En este trabajo se propone diseñar e implementar un nuevo backend para Controllers que soporte las unidades de procesamiento CPUs donde los kernels van a ser ejecutados.

1.3. Objetivos

Este trabajo propone una extensión del modelo de programación Controllers [2, 4] que permita utilizar un grupo de núcleos de una CPU como un acelerador hardware permitiendo alta portabilidad para utilizar otros aceleradores como GPUs o FPGAs de una forma eficiente con cambios mínimos en el código. Además este trabajo debe ajustar la granularidad en la paralelización de los kernels de forma automática, ya que en otros aceleradores como GPUs o FPGAs se usa un paralelismo de grano muy fino dado que tienen muchos núcleos. No obstante, usar un paralelismo de grano muy fino no es recomendable en una CPU debido al enorme sobrecoste que causaría crear todos esos hilos. Por lo tanto, proponemos este modelo en el que el código de los kernels programado por los usuarios de la forma típica para otros aceleradores de hardware se traduce de forma sistemática en tiempo de compilación.

Este modelo debe dar flexibilidad al programador en la gestión de la memoria permitiéndole seleccionar si se debe usar la memoria del grupo de núcleos como memoria separada del resto de la CPU y realizar copias o que los accesos desde los kernels sean "copias cero" accediendo directamente a la memoria del device. También debe permitir la detección automática de dependencias para solapar tareas cuando sea posible, incrementando así el rendimiento.

Los objetivos específicos de este trabajo son los siguientes:

1. Estudiar y comprender las librerías Hitmap [1] y Controllers [2, 4] sobre las que se asienta este trabajo.
2. Diseñar e implementar un nuevo backend de Controllers para dispositivos CPU.
3. Diseñar, implementar y ejecutar benchmarks que permitan validar el modelo.
4. Obtención y análisis de resultados.
5. Obtención de conclusiones.

1.4. Planificación y presupuesto del proyecto

1.4.1. Planificación del proyecto

En la tabla 1.1 se muestra una descomposición del trabajo realizado durante el desarrollo de este proyecto en tareas.

Este proyecto parte de las librerías Controllers y Hitmap. Son dos herramientas bastante complejas. Sin embargo, este trabajo resultará en una expansión de Controllers mientras que Hitmap es solo una herramienta auxiliar, de la cual solo se requieren algunas características. Esto implica que se requiere tener una comprensión del funcionamiento interno de Controllers, pero no es necesario alcanzar el mismo nivel de comprensión de Hitmap. Bastará con conocer la interfaz de usuario y algunos detalles de implementación de esta librería.

El proceso de implementación es una de las partes en la que más retrasos pueden ocurrir debido a la complejidad conceptual asociada con la meta-programación. El programador debe conocer los fragmentos de código que realmente se expandirán en el programa real a partir de los macros del preprocesador utilizados. La compilación condicional y la ayuda limitada del compilador durante la depuración convierten esto en una tarea complicada. Esta tarea de implementación es especialmente delicada cuando se realizan cambios a la parte general de Controllers, ya que esto afecta a todos los backends y es necesario tener en cuenta el comportamiento de las tecnologías que se utilizan en todos los backends.

La tarea de experimentación no debería ser una tarea excesivamente compleja una vez que el backend de la CPU esté integrado correctamente en los controladores. Aun así, debemos tener en cuenta que este es el primer contacto del estudiante con una tarea de este tipo y el formato de los resultados es algo distinto al de otros backends lo que implica modificar los scripts usados para agrupar y mostrar los resultados.

Rol	Tarea	Tiempo (horas)
Estudiante	Entender Hitmap	10
	Entender Controllers	100
	Lectura	50
	Implementación del backend	140
	Experimentación	50
	Escribir memoria	60
	Otros	40
Tutores	Reuniones	40
	Correcciones a la memoria	10

Cuadro 1.1: Estimación del tiempo invertido en el proyecto.

1.4.2. Presupuesto

El presupuesto para el proyecto se distribuye de la siguiente manera:

Gastos de amortización de maquinas:

- Hydra: Esta máquina costó aproximadamente 10000€. El tiempo estimado de vida de está maquina es de 4 años, lo que nos deja un coste de 0.28€ por hora de uso. Esta máquina se planea usar para realizar la experimentación. Esta tarea se estima que requiera unas 250 horas de uso, esto nos deja un coste de 70€
- Manticore: Esta máquina costó aproximadamente 38000€. El tiempo estimado de vida de está maquina es de 4 años, lo que nos deja un coste de 1.08€ por hora de uso. Esta máquina se usa durante el proceso de implementación. Esta tarea se estima que requiera unas 20 horas de uso, esto nos deja un coste de 21.6€
- Máquina del desarrollador: la máquina utilizada tuvo un coste de 1400€, con vistas a ser utilizada durante 6 años. El coste por hora es de 0.03€. Esta maquina se usará durante todo el tiempo que dure el proyecto, 450 horas. Lo que nos deja un coste de 13.5€.

Coste de personal:

- Sueldo de ingeniero junior: estimamos el sueldo de un ingeniero informático junior en unos 50000€ brutos al año, y el número de días laborables en un año en alrededor de 250. Si se supone una jornada laboral de 8 horas, llegamos a un coste de 25€ por una hora de trabajo del desarrollador.

  Estimamos la cantidad de horas de trabajo en 450 horas. Lo que nos da un coste de 11250€

- Sueldo de los Tutores: estimamos el sueldo de los tutores en unos 100000€ brutos al año. Realizando el cálculo anterior, tenemos un coste de 50€ por cada hora de consultoría.

  Se estiman 20 reuniones semanales de 1 hora con los 2 tutores presentes, 10 horas de revisiones de la memoria y 15 horas empleadas en la resolución de dudas mediante correo electrónico. Obtenemos un total de 65 horas lo que corresponde a un coste de 3250€

Costes de software: en este proyecto solo se ha trabajado con software de uso libre, por lo tanto no hay necesidad de pagar licencias. Así que el coste es 0€.

Sumando todos estos costes obtenemos que el coste total del proyecto se estima en 14605.1€.

1.4.3. Plan de contingencia

La tabla 1.2 describe un plan de contingencia para los sucesos que pueden ocurrir a lo largo del desarrollo del proyecto.

Suceso	Plan de contingencia
Avería de la máquina del cluster usada para el desarrollo (Manticore)	Se continua el desarrollo en Hydra, otra de las máquinas del cluster
Avería del cluster durante el desarrollo	Se continua el desarrollo en la maquina personal del desarrollador
Avería del cluster durante el proceso de experimentación	Uso de los recursos de otro centro de supercomputación accesible para el grupo Trasgo para realizar las pruebas
Retraso en la implementación del backend de CPU	Se retrasa la experimentación
Pérdida de las aplicaciones almacenadas en la máquina usada para el desarrollo por un fallo de almacenamiento	Se realizarán backups semanales en el repositorio de gitlab del grupo Trasgo
No se alcanza la fecha de entrega del proyecto por causas de fuerza mayor	Se realiza la entrega del proyecto en segunda convocatoria

Cuadro 1.2: Plan de contingencia del proyecto.

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 Controllers al inicio del proyecto. El capítulo 3 describe la especificación del modelo propuesto para el backend de CPU de Controllers. El capítulo 4 describe la implementación realizada del modelo sobre la librería Controllers. El capítulo 5

expone los resultados experimentales obtenidos en los casos de estudio que permiten validar el modelo propuesto. Por ultimo, el capítulo 6 resume las conclusiones obtenidas y objetivos cumplidos.

Capítulo 2

Estado del arte y conceptos previos

En este capítulo se introducen los siguientes conceptos:

- Las arquitecturas NUMA de CPU.
- La interfaz de programación OpenMP y la librería hwloc usadas en este trabajo.
- Las librerías Hitmap y Controllers, desarrolladas por el Grupo Trasgo.

2.1. Arquitecturas NUMA

En los sistemas UMA (Uniform Memory Access) [16], las CPU se conectan a través de un bus del sistema (bus frontal) al Northbridge. El Northbridge contiene el controlador de memoria y, por lo tanto, todas las comunicaciones con la memoria deben pasar por él. El controlador de E / S, responsable de administrar las E / S de todos los dispositivos, está conectado al Northbridge. Por lo tanto, cada E / S tiene que pasar por Northbridge para llegar a la CPU. Sin embargo, debido al ancho de banda interno de Northbridge y la naturaleza de transmisión de los primeros protocolos de caché snoopy, se consideró que UMA tenía una escalabilidad limitada.

Para mejorar el rendimiento de la interfaz de memoria y suministrar datos a todos los núcleos, los diseños de procesadores más nuevos integran el controlador de memoria en el chip con el procesador. Así surgen la arquitectura NUMA (Non-Uniform Memory Access) [14, 15]. Esta arquitectura proporciona un acceso con mayor ancho de banda y menor latencia a la memoria local. El acceso a la memoria remota, que generalmente es local para otros procesadores en el host, es posible, pero más lento.

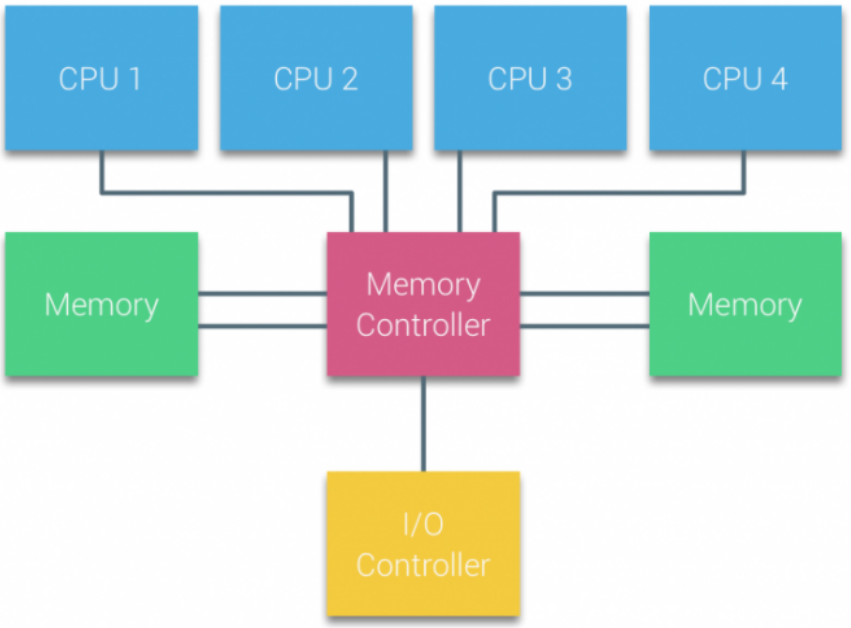


Figura 2.1: Arquitectura UMA - Uniform Memory Access [16].

El controlador de memoria en chip también tiene otra ventaja. En un multiprocesador multinúcleo (un multiprocesador construido a partir de varios procesadores multinúcleo), un controlador de memoria local permite mayor escalabilidad del sistema ya que no hay un solo controlador de memoria central. En cambio, el espacio de direcciones físicas se divide entre procesadores y los núcleos de cada procesador pueden acceder directamente solo a una parte de la memoria física a través de la interfaz de memoria local. Para admitir el modelo familiar de un multiprocesador de memoria compartida, cada procesador (y sus núcleos) debe poder acceder no solo a la memoria local conectada directamente, sino también a la memoria local de otros procesadores. Estos accesos remotos a la memoria pasan a través de una interconexión entre chips que conecta los procesadores.

El rendimiento de la interconexión entre chips, sin embargo, es menor que el rendimiento de los controladores de memoria en el chip. Los accesos a memoria remota que pasan a través de la interconexión entre chips también encuentran latencias mayores que las latencias de los accesos a la memoria local. La penalización del rendimiento de los accesos remotos a la memoria es significativa (a esta penalización le llamamos factor NUMA); en las implementaciones actuales, el factor NUMA puede ser tan alto como 2 (equivalente a una ralentización 2X) para algunas aplicaciones [14]. La optimización de rendimiento más importante para los sistemas NUMA considerada hasta ahora es aumentar la localidad de datos del sistema, es decir, asignar memoria cerca de los cálculos que acceden a ella, de modo que se reduzca el número de accesos remotos y, por lo tanto, la penalización de rendimiento específica de NUMA se evita.

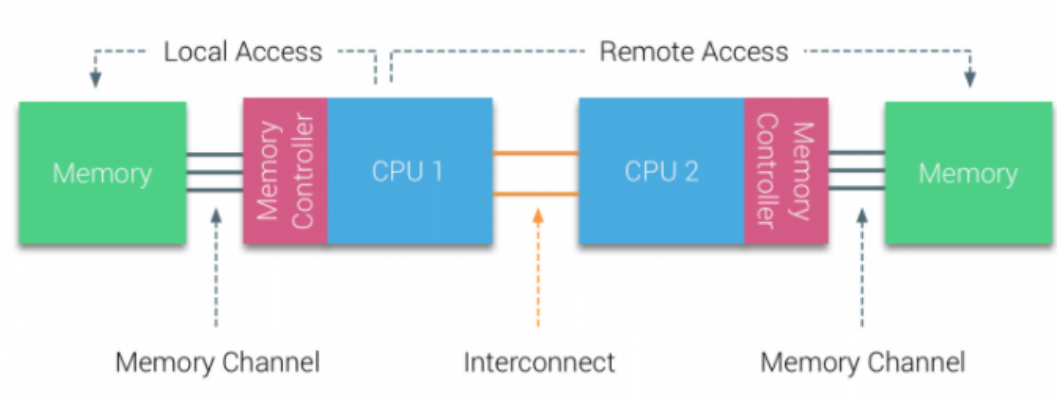


Figura 2.2: Arquitectura NUMA - Non-Uniform Memory Access [16].

2.2. Interfaz de programación OpenMP

OpenMP [9] es una interfaz de programación de aplicaciones (API) de memoria compartida cuyas características se basan en esfuerzos previos para facilitar la programación paralela en sistemas de memoria compartida. Es un acuerdo alcanzado entre los miembros de la ARB, que comparten un interés en un enfoque portátil, fácil de usar y eficiente para la programación paralela de memoria compartida. OpenMP está diseñado para ser adecuado para su implementación en una amplia gama de Arquitecturas SMP (*Symmetric Multi-Processing*). OpenMP no es un nuevo lenguaje de programación. Más bien, es es una notación que se puede agregar a un programa secuencial en Fortran, C o C ++ para describir como se debe repartir el trabajo entre los hilos que se ejecutarán en los diferentes núcleos y para acceder a la memoria compartida.

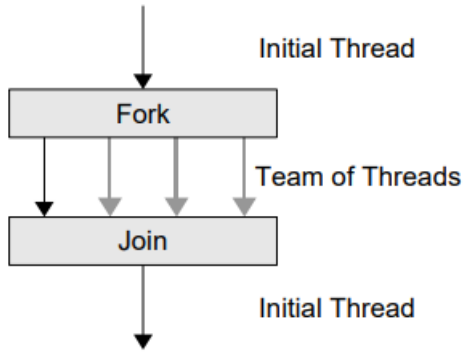


Figura 2.3: **El modelo de programación fork-join soportado por OpenMP:** el programa comienza como un único hilo de ejecución, el hilo inicial. Un equipo de hilos se bifurca al comienzo de una región paralela y se une al final [9].

OpenMP implementa el modelo de programación “fork-join” [11], que se ilustra en la

figura 2.3. Bajo este enfoque, el programa comienza como un único hilo de ejecución, como un programa secuencial. El hilo que ejecuta este código se denomina hilo inicial. Siempre que un subproceso encuentra una la directiva parallel de OpenMP mientras está ejecutando el programa, crea un equipo de hilos (esto es el fork), se convierte en el maestro del equipo y colabora con los demás miembros del equipo para ejecutar el código marcado por la directiva. Al final de la directiva, solo el hilo original, o maestro del equipo, continúa; todos los otros finalizan (esto es el join). Cada porción de código encerrada por una directiva parallel se llama región paralela.

La API de OpenMP comprende un conjunto de directivas de compilación, rutinas de biblioteca en tiempo de ejecución y variables de entorno para especificar el paralelismo de memoria compartida en programas Fortran y C / C ++. Una directiva OpenMP es un comentario o pragma con un formato especial que generalmente se aplica al código ejecutable que lo sigue inmediatamente en el programa. Una directiva o una rutina de OpenMP generalmente afecta solo a los hilos que lo encuentran. Muchas de las directivas se aplican a un *bloque estructurado* de código.

OpenMP proporciona medios para que el usuario pueda:

- crear equipos de hilos para ejecución paralela,
- especificar cómo compartir el trabajo entre los miembros de un equipo,
- declarar variables tanto compartidas como privadas, y
- sincronizar hilos y permitirles realizar determinadas operaciones de forma exclusiva (es decir, sin interferencia de otros hilos).

2.2.1. Directivas y características de OpenMP

En esta sección se explicarán algunas de las principales directivas de OpenMP que se han usado en este trabajo:

Directiva Parallel

```
#pragma omp parallel [clause[[,] clause]...]
    structured block
```

Figura 2.4: **Sintaxis de la directiva parallel en C / C ++** - La región paralela implícitamente termina al final del bloque estructurado. Esta es una llave de cierre (}) en la mayoría de casos.

La directiva parallel es crucial en OpenMP: un programa sin una construcción paralela se ejecutará secuencialmente. Su sintaxis C / C ++ se muestra en la Figura 2.4. Esta construcción se utiliza para especificar los cálculos que deben ejecutarse en paralelo. Las

partes del programa que no están incluidas en una construcción paralela se ejecutarán en serie. Cuando un hilo encuentra esta construcción, se crea un equipo de hilos para ejecutar la región paralela asociada, que es el código contenido dinámicamente dentro de la construcción paralela. Pero aunque esta construcción asegura que los cálculos se realicen en paralelo, no distribuye el trabajo de la región entre los hilos de un equipo. De hecho, si el programador no usa la sintaxis adecuada para especificar esta acción, el trabajo se replicará. Al final de una región paralela, hay una barrera implícita que obliga a todos los hilos a esperar hasta que se complete el trabajo dentro de la región. Solo el hilo inicial continúa la ejecución después del final de la región paralela.

El hilo que se encuentra con la construcción paralela se convierte en el *master* del nuevo equipo. A cada hilo del equipo se le asigna un número de hilo único (también denominado "ID de hilo") para identificarlo. Van desde cero (para el hilo maestro) hasta uno menos que el número de hilos dentro del equipo, y el programador puede acceder a ellos mediante la función `omp_get_thread_num()`. Aunque la región paralela es ejecutada por todos los hilos del equipo, cada hilo puede seguir una ruta de ejecución diferente. Una forma de lograr esto es usando los ID de hilo.

Si un hilo en un equipo que ejecuta una región paralela encuentra otra construcción parallel, crea un nuevo equipo y se convierte en el master de ese nuevo equipo. Esto generalmente se conoce en OpenMP como "paralelismo anidado".

Directiva For

#pragma omp for *[clause[[,] clause]...]*
`for-loop`

Figura 2.5: **Sintaxis de la directiva for en C / C ++**

La directiva for hace que las iteraciones del bucle a continuación se ejecuten en paralelo. Las iteraciones de bucle se distribuyen entre los hilos en tiempo de ejecución. Esta es probablemente la más utilizada de las funciones de trabajo compartido. Su sintaxis para C / C ++ se muestra en la Figura 2.5.

En los programas C y C ++, el uso de esta construcción se limita a los tipos de bucles en los que se puede contar el número de iteraciones; es decir, el bucle debe tener una variable contador entera cuyo valor se incrementa (o decrementa) en una cantidad fija en cada iteración hasta que se alcanza algún límite superior (o inferior) especificado.

Directiva Barrier

Una barrera es un punto en la ejecución de un programa donde los hilos se esperan unos a otros: ningún hilo en el equipo de hilos al que se aplica puede avanzar más allá de una barrera hasta que todos los hilos del equipo hayan llegado a ese punto. Muchas construcciones de OpenMP tienen una barrera implícita. Es decir, el compilador inserta automáticamente

```
#pragma omp barrier
```

Figura 2.6: **Sintaxis de la directiva barrier en C / C ++** - Esta directiva aplica a la región paralela más interna que la encierra.

una barrera al final de la construcción, de modo que todos los hilos esperan allí hasta que se haya completado todo el trabajo asociado con la construcción. Por lo tanto, a menudo es innecesario que el programador agregue explícitamente una barrera a un código. Sin embargo, si se necesita uno, OpenMP proporciona una construcción que lo hace posible. La sintaxis en C / C ++ se muestra en la Figura 2.6.

Directiva Critical

```
#pragma omp critical [(name)]
    structured block
```

Figura 2.7: **Sintaxis de la directiva crítical en C / C ++:** el bloque estructurado es ejecutado por todos los subprocesos, pero solo uno a la vez ejecuta el bloque. Opcionalmente, la construcción puede tener un nombre.

La directiva crítical proporciona un medio para garantizar que varios hilos no intenten actualizar los mismos datos compartidos simultáneamente. El código asociado se denomina región crítica o sección crítica. Opcionalmente, se le puede dar un nombre a una construcción crítica. Cuando un hilo encuentra una construcción crítica, espera hasta que ningún otro hilo esté ejecutando una región crítica con el mismo nombre. En otras palabras, nunca existe el riesgo de que varios hilos ejecuten el código contenido en la misma región crítica al mismo tiempo. La sintaxis de la directiva crítical en C / C ++ se muestra en la Figura 2.7.

Directiva Atomic

```
#pragma omp atomic
    statement
```

Figura 2.8: **Sintaxis de la directiva atomic en C / C ++:** la instrucción es ejecutada por todos los hilos, pero solo un hilo a la vez ejecuta la instrucción.

La directiva atomic, que también permite que varios hilos actualicen los datos compartidos sin interferencias, puede ser una alternativa eficiente a la región crítica. A diferencia de otras construcciones, se aplica solo a la expresión que la sigue inmediatamente; esta declaración debe tener cierta forma para que la construcción sea válida y, por lo tanto, su rango de aplicabilidad está estrictamente limitado. La sintaxis para C / C ++ se muestra en la Figura 2.8.

La directiva atomic permite la actualización eficiente de variables compartidas por múltiples hilos en plataformas de hardware que soportan operaciones atómicas. La razón por la que se aplica a una sola declaración de asignación es que protege las actualizaciones de una ubicación de memoria individual, la que está en el lado izquierdo de la asignación. Si el hardware admite instrucciones que leen desde una ubicación de memoria, modifican el valor y vuelven a escribir en la ubicación en una sola acción, entonces atomic indica al compilador que utilice dicha operación. Si un hilo está actualizando atómicamente un valor, ningún otro hilo puede hacerlo simultáneamente. Esta restricción se aplica a todos los hilos que ejecutan un programa, no solo a los subprocesos del mismo equipo. Sin embargo, para garantizar esto, el programador debe marcar todas las actualizaciones potencialmente simultáneas en una ubicación de memoria mediante esta directiva.

Locks

```
void omp_func_lock (omp_lock_t *lck)
```

Figura 2.9: **Sintaxis general de las rutinas de locks en C / C ++:** para una rutina específica, func expresa su funcionalidad; func puede asumir los valores init, destroy, set, unset, test. Los valores de las cerraduras anidadas son init nest, destroy nest, set nest, unset nest, test nest.

Además de las directivas de sincronización presentadas anteriormente, la API de OpenMP proporciona un conjunto de rutinas de biblioteca en tiempo de ejecución de locks de propósito general y bajo nivel, similar en función al uso de semáforos. Estas rutinas proporcionan una mayor flexibilidad para la sincronización que el uso de secciones críticas o construcciones atómicas. La sintaxis general de las rutinas de la biblioteca de locks se muestra en la Figura 2.9.

Las rutinas operan en variables de bloqueo de propósito especial, a las que se debe acceder solo a través de las rutinas de bloqueo. Hay dos tipos de locks: locks simples, que pueden no estar bloqueados si ya están en un estado bloqueado, y locks "nested", que pueden bloquearse varias veces por el mismo hilo. Las variables de bloqueo simple se declaran con el tipo especial `omp_lock_t` en C / C ++. Las variables de bloqueo anidables se declaran con el tipo especial `omp_nest_lock_t` en C / C ++. En C, las rutinas de bloqueo necesitan un argumento que sea un puntero a una variable de bloqueo del tipo apropiado. El procedimiento general para usar locks es el siguiente:

1. Definir las variables de bloqueo (simples o anidadas).
2. Inicialice el bloqueo mediante una llamada a `omp_init_lock`.
3. Bloquear la lock utilizando `omp_set_lock` o `omp_test_lock`. Este último comprueba si la lock está realmente disponible antes de intentar bloquearla.
4. Desbloquear la lock después de que el trabajo haya terminado mediante una llamada a `omp_unset_lock`.

5. Destruir la lock mediante una llamada a `omp_destroy_lock`.

Una restricción importante del sistema de locks de OpenMP es el concepto de *propiedad de una lock*. Una lock que no está desbloqueada no tiene propietario y se convierte en propiedad de la *task region* que llama a `omp_set_lock`. Una lock bloqueada **solo** puede ser desbloqueada por la region de tarea que tenga la propiedad de esa lock.

Directiva Task

Cuando un hilo encuentra una directiva task, se genera una tarea explícita a partir del código del bloque estructurado asociado. El entorno de datos de la tarea se crea de acuerdo con las cláusulas de atributos de intercambio de datos en la construcción de la tarea. El entorno de datos de la tarea se destruye cuando se completa el código de ejecución del bloque estructurado asociado. El hilo que se encuentra la directiva puede ejecutar inmediatamente la tarea o aplazar su ejecución. En el último caso, se puede asignar la tarea a cualquier hilo del equipo. La finalización de la tarea se puede garantizar mediante construcciones de sincronización de tareas.

Una clausula a destacar en esta directiva es la clausula *depend* que permite marcar dependencias entre las tareas para asegurar que se ejecutan en un orden concreto.

2.3. Librería Hwloc

El paquete de software Portable Hardware Locality (hwloc) [10] proporciona una abstracción portátil (entre sistemas operativos, versiones, arquitecturas, ...) de la topología jerárquica de las arquitecturas modernas, incluidos los nodos de memoria NUMA, sockets, cachés compartidas, núcleos y SMT. También recopila varios atributos del sistema, como información de memoria y caché, así como la ubicación de los dispositivos de E/S, como interfaces de red, InfiniBand HCAs o GPUs.

Hwloc tiene como principal objetivo ayudar a las aplicaciones a recopilar información sobre plataformas informáticas paralelas cada vez más complejas para explotarlas de manera adecuada y eficiente. Por ejemplo, dos tareas que cooperan estrechamente probablemente deberían colocarse en núcleos que comparten una caché. Sin embargo, es mejor distribuir dos tareas independientes que consumen mucha memoria en diferentes sockets para maximizar su rendimiento de memoria.

Hardware Locality (hwloc) se diseñó a partir de la idea de que las arquitecturas actuales y de próxima generación son jerárquicas. De hecho, las máquinas actuales constan de varios sockets de procesador que contienen varios núcleos compuestos por uno o varios hilos de hardware. Esto llevó a representar la arquitectura de hardware como un árbol de recursos. La Figura 2.10 muestra la vista jerárquica correspondiente (incluido el conocimiento de las cachés compartidas) para un host de cuatro núcleos y dos sockets.

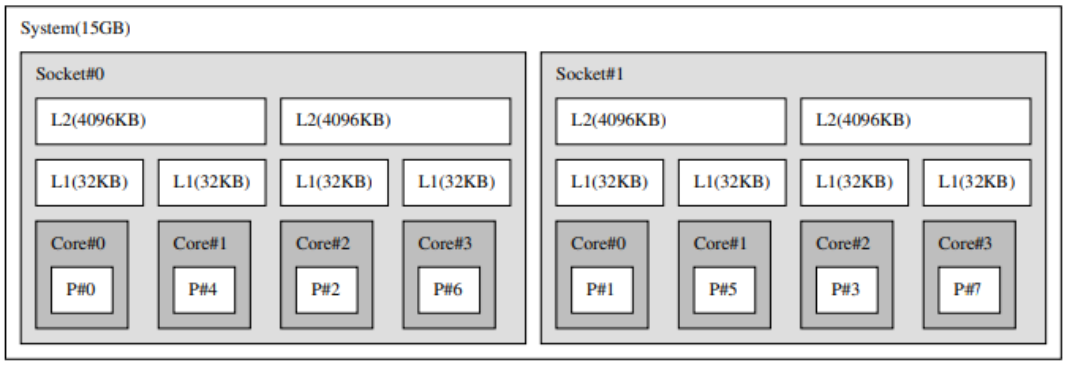


Figura 2.10: Salida gráfica de la herramienta *lstopo* de un quad-core con 2 sockets [10].

Una vez que la aplicación o el sistema de tiempo de ejecución ha encontrado los objetos interesantes en el árbol de topología, puede recuperar información de sus atributos para adaptar su comportamiento a las características subyacentes del hardware. También es posible vincular subprocesos o procesos a cualquier objeto utilizando la API de hwloc. Para hacerlo, hwloc usa su propia estructura *Cpuset* que contiene la máscara de bits de los procesadores lógicos permitidos. Cada objeto del árbol contiene su propio *Cpuset* que se puede modificar o combinar con otros objetos a través de un extenso conjunto de operaciones.

2.4. Herramientas desarrolladas por el grupo Trasgo

En esta sección se describen las librerías desarrolladas por el grupo de investigación Trasgo[12].

2.4.1. Librería Hitmap

La librería Hitmap [1] ofrece técnicas de particionado y mapeo automático de datos, de forma eficiente y configurable, en tiempo de ejecución. Esta librería define una interfaz de abstracción y un sistema de plug-in que encapsula técnicas regulares e irregulares, ayudando a generar código independientemente de la función de mapeo seleccionado. Hitmap soporta el particionado (tiling) de array distribuciones dispersas o compactas, este particionado permite la implementación de paralelismo de datos y tareas según el modelo SPMD. Esta librería ofrece funcionalidades para crear, manipular, distribuir y comunicar estas particiones (tiles) y jerarquía de particiones.

Conceptos

La librería Hitmap introduce los siguientes conceptos:

- *Signature:* Una *signature* S se define como una terna que representa un subespacio de índices sobre un array unidimensional. Una signature sigue la notación de Fortran90 o MATLAB de selección de índices de un array. La cardinalidad de una signature es el número de índices diferentes del dominio.

$$S \in Signature = (begin : end : stride)$$

$$Card(s \in Signature) = b(s.end - s.begin)/s.stridec$$

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

$$h \in Shape = (S_0, S_1, S_2, ..., S_{n-1})$$

$$Card(h \in Shape) = \prod_{i=0}^{n-1} Card(S_i)$$

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

$$Tile_{h \in Shape} : (S_0 \times S_1 \times S_2 \times ...S_{n-1}) \rightarrow < type >$$

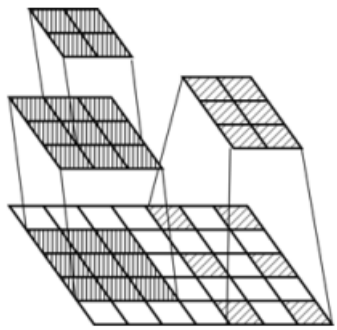


Figura 2.11: Creación de tiles de un array original [1]

Funcionalidades

Hitmap ofrece tres conjuntos de funcionalidades:

- *Funciones de tiling:* Definición y manipulación de arrays y tiles. Estas funciones pueden utilizarse de forma independiente. Estas funciones mejoran el rendimiento al acceder a los datos en código secuencial al igual que al generar manualmente distribuciones de datos para ejecuciones paralelas. La figura 2.11 muestra un ejemplo de jerarquía de tiles.

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

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

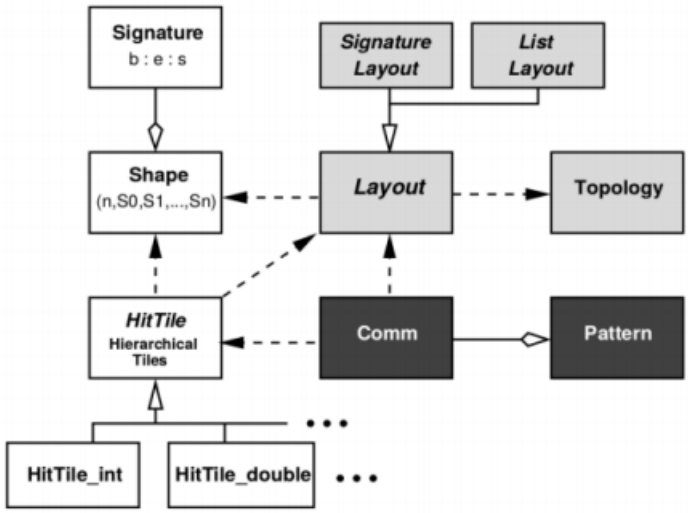


Figura 2.12: Diagrama UML de la arquitectura de la librería Hitmap [1].

2.4.2. Controllers

Controllers [2, 4] es un modelo de programación paralela que propone un objeto abstracto para coordinar las actividades de ejecución y gestión de memoria en un acelerador o un conjunto de núcleos de CPU. Un objeto controlador está asociado a una instancia particular de un dispositivo durante su creación.

Controllers ofrece características como:

- Mecanismo para definir diferentes implementaciones de un kernel bajo un mismo nombre; desde kernels comunes que pueden ser reutilizados por distintos tipos de máquinas, hasta kernels específicos programados para un grupo concreto de dispositivos.

- Mecanismo transparente para administración de memoria, incluyendo comunicaciones de estructuras de datos entre el computador principal y sus correspondientes copias en las aceleradoras hardware.

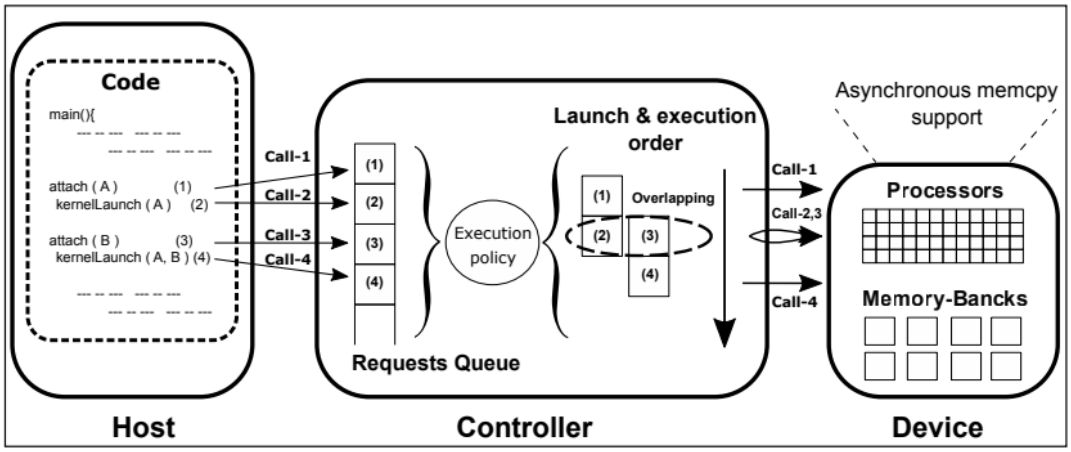


Figura 2.13: Diagrama del modelo de arquitectura de Controllers. Las peticiones asíncronas son añadidas a una cola. La entidad del Controlador será la encargada de la ejecución de estas transferencias y las ejecuciones de los kernels.

- Sistema de mejora para seleccionar la configuración óptima en el lanzamiento de kernels (como la geometría de los bloques de hilos), guiada por una caracterización del kernel dada por el usuario con una sintaxis simple en la definición del kernel.

Declaración de kernels

En el modelo de Controllers, un kernel se declara utilizando dos primitivas. La primera es `CTRL_KERNEL_PROTO`, que declara un prototipo para todas las implementaciones de un kernel dado. Vea las líneas 8-13 en la figura 2.1. Declara el número de implementaciones disponible y para qué backends o dispositivos están diseñadas esas implementaciones. El *Controller* lo utiliza para localizar, en tiempo de ejecución, la mejor implementación disponible de un kernel para el dispositivo asociado. En el ejemplo, solo hay una implementación declarada.

La palabra clave GENERIC indica que se puede usar en cualquier backend. Otros nombres de palabras clave están asociados a los tipos específicos de backend o dispositivo. El resto del prototipo es la descripción de los parámetros del kernel, incluidos sus roles de entrada / salida, tipos y nombres. `IVAL` indica un parámetro de valor. `IN`, `OUT` o `IO` (in / out) indica el rol de un HitTile recibido como referencia.

Cada implementación de kernel se declara utilizando el primitivo `CTRL_KERNEL`. Ver líneas 1-5 en la figura 2.1. Los primeros parámetros son el nombre del kernel y el tipo de implementación. Después de la declaración 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.

```
CTRL_KERNEL(Add, GENERIC, KHitTile_float A, KHitTile_float B, KHitTile_float C
    ,
{
  hit(C, 2, thread_id.x, thread_id.y) =
      hit(A, 2, thread_id.x, thread_id.y) +
      hit(B, 2, thread_id.x, thread_id.y);
});

CTRL_KERNEL_PROTO( Add,
    1, GENERIC,
    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 Controllers.

Declaración de tareas de host

El modelo de Controllers también permite la declaración de tareas de host que se ejecutarán asíncronamente en el host. 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 no se indica la implementación.

Operaciones de Controllers

En Controllers las interacciones entre el host y el dispositivo se interpretan como una serie de peticiones controladas por el código coordinador ejecutado en el host. Una petición puede ser uno de los siguientes tipos:

- `Ctrl_Alloc(ctrl, tile1, tile2...)`. Una petición para acomodar una estructura de datos en el host y el dispositivo.
- `Ctrl_Free(ctrl, tile1, tile2...)`. Una petición para deallocate the memory image/s of a data structure.
- `Ctrl_MoveTo(ctrl, tile1, tile2...)` Una petición para transferir los valores de las estructuras de datos del espacio de memoria del host al espacio de memoria del dispositivo.
- `Ctrl_MoveFrom(ctrl, tile1, tile2...)` Una petición para transferir los valores de las estructuras de datos del espacio de memoria del dispositivo al espacio de memoria del host.
- `Ctrl_Launch(ctrl, f, threads, params`. Una petición para ejecutar un kernel en el dispositivo. Recibe un nombre de función *f*, un bloque de hilos *threads* con los que ejecutar el kernel y los argumentos que recibe la función *f*.

- `Ctrl_HostTask(ctrl, h, params`. Una petición para ejecutar la función h en el host.
- `Ctrl_WaitTile(ctrl, tile1, tile2...)`. Una petición para block the execution of the host code until all requests involving each tile have finished. In many programming models, such as CUDA, this operation is implicit by default after blocking DTH(x) requests, although it can be relaxed by the programmer using explicit asynchronous transfers. In our asynchronous model, this operation is explicitly used only when the coordination code needs to use values retrieved from the device. For example, this operation would be necessary in a loop with a convergence condition that is calculated with a reduction operation in the device.
- `Ctrl_GlobalSync(ctrl)` a global wait request, that blocks the execution until all previous requests of any type finish.

En la figura 2.2 se puede ver un ejemplo del main de un código de suma de matrices usando la librería Controllers.

Hilos internos y ctrl_block

En un programa de Controllers todas las interacciones explicadas en la seccion anterior se encuentran en un bloque estructurado precedido por `__ctrl_block__(num_ctrl)`. El parámetro *num_ctrl* indica el numero de controladores que se utilizarán en el bloque, aunque actualmente el prototipo de Controllers solo permite un controlador por bloque.

Esta estructura se utiliza para crear y preparar los hilos internos utilizados por los controladores. Los hilos empleados internamente son los siguientes:

- Hilo principal: es el hilo ya existente, esté hilo introduce las tareas marcadas por el programador en la cola del controlador correspondiente.
- Hilo de host: este hilo se ocupa de la ejecución de todas las tareas de host que se lancen en el bloque.
- Hilo gestor de cola o hilo de controlador: cada controlador tiene su propio hilo de gestión de cola. Este hilo se ocupa de extraer las tareas de la cola del controlador y lanzarlos al acelerador teniendo en cuenta la política seleccionada y las dependencias de esa tarea especifica.

```
int main(int argc, char *argv[]){
    ...
    //Logical threads for the device
    Ctrl_Thread threads;
    Ctrl_ThreadInit(threads, SIZE_1, SIZE_2);
    __ctrl_block__(1)
    {
        //Crear controlador
        PCtrl ctrl = Ctrl_Create(CTRL_TYPE_CUDA, CTRL_POLICY_ASYNC, DEVICE,
    NULL);
        //Crear estructuras de datos
        HitTile_float matrixA = Ctrl_Alloc(ctrl, float, hitNewShapeSize(SIZE_1
    , SIZE_2) );
        HitTile_float matrixB = Ctrl_Alloc(ctrl, float, hitNewShapeSize(SIZE_1
    , SIZE_2) );
        HitTile_float matrixC = Ctrl_Alloc(ctrl, float, hitNewShapeSize(SIZE_1
    , SIZE_2) );

        //Inicializar en el host
        Ctrl_HostTask(ctrl, Init_Tiles, matrixA, matrixB, matrixC);

        //sincronizar y comenzar el reloj
        Ctrl_GlobalSync(ctrl);
        exec_clock = omp_get_wtime();


        //Modo explicito: Ctrl_MoveTo(ctrl, matrixA, matrixB, matrixC);

        //Launch kernel operation
        Ctrl_Launch(ctrl, Add, threads, matrixA, matrixB, matrixC);

        //Modo explicito: Ctrl_MoveFrom(ctrl, matrixC);

        //Sincronizar y parar el reloj
        Ctrl_GlobalSync(ctrl);
        exec_clock = omp_get_wtime() - exec_clock2;

        //Calcular la norma en el host
        Ctrl_HostTask(ctrl, Norm_calc, matrixC);

        Ctrl_Free(ctrl, matrixA, matrixB, matrixC);
        Ctrl_Destroy(ctrl);
    } //__ctrl_block__
    ...
}//main
```

Listing 2.2: Ejemplo del código del main de una suma de matrices usando la librería Controllers. Las operaciones de movimientos de memoria que se realizan implícitamente se muestran con comentarios como se realizarían en el modo explicito.

Capítulo 3

Descripción de la solución

En este capítulo se describen los siguientes elementos:

- Descripción conceptual del backend de Controllers para CPUs
- Propuestas para el nuevo sistema de creación de hilos de Controllers
- Aproximaciones para el lanzamiento y sincronización de tareas en CPU

3.1. Propuesta de solución

En este trabajo se propone el diseño y la implementación de un nuevo backend de Controllers [2] que permita utilizar un grupo de núcleos de CPU seleccionados en tiempo de ejecución, como un acelerador hardware externo, permitiendo lanzar kernels de forma asíncrona con una gestión de dependencias automática que permita el solapamiento de tareas cuando sea posible.

3.2. Construcción de hilos en Controllers

En esta sección se exploran las diferentes propuestas consideradas para adaptar la creación de los hilos internos de Controllers a las necesidades del nuevo backend de CPUs así como prepararlo para la futura versión en la que se permitan múltiples controladores dentro del mismo `__Ctrl_block__()`.

Estas propuestas deben permitir la existencia de controladores que requieran diferente número de hilos entre controladores diferentes. Éste como es el caso de los controladores de CPU que requieren un hilo extra para ejecutar los kernels. En adelante, a estos hilos que solo

hace falta crear para algunos tipos de controladores los denominaremos *hilos extra*. En la versión actual de Controllers solo los controladores de tipo CPU requieren hilos extra, pero la solución debe estar abierta a nuevos backends que puedan requerir hilos extra.

3.2.1. Primera opción: Un hilo por Ctrl

La primera propuesta se basa en crear el hilo principal, el hilo de host y un hilo para cada controlador en el macro `__Ctrl_block__()` y posteriormente en la creación del controlador crear el resto de hilos en caso de que sean necesarios a partir del hilo creado para ese controlador. En la figura 3.1a se muestra de forma conceptual la creación de los hilos en un ejemplo con 2 controladores.

El problema con esta aproximación es que, para el futuro de Controllers una de las necesidades para la versión de múltiples controladores es eliminar el hilo gestor de cola asociado a los controladores, por lo tanto en esta futura versión, habría controladores que no requieran ningún hilo, como se muestra en la figura 3.1b y crear hilos inútiles no es razonable.

3.2.2. Segunda opción: Un hilo por Ctrl que necesita hilos extra

Esta segunda opción surge como solución al problema de la anterior. La solución para no tener los hilos inútiles asociados a los controladores que no requieren ningún hilo extra es añadir un nuevo parámetro al macro `__Ctrl_block__()` que represente el número de controladores que necesitarán hilos extra. De esta forma podemos crear un hilo para cada controlador que requiera hilos extra y solo para esos controladores, evitando así la creación de hilos innecesarios.

El problema con esta aproximación es que requiere modificar la interfaz de `__Ctrl_block__()` añadiendo un parámetro que indique el número de controladores que van a requerir hilos.

3.2.3. Tercera opción: Todos los hilos salen del hilo de host task al principio

En esta aproximación en el macro `__Ctrl_block__()` se crean únicamente el hilo principal y el hilo de host. Posteriormente, tras la creación de todos los controles ya conocemos sus tipos y por lo tanto cuantos hilos requieren, así que dividimos este hilo de host para generar tantos hilos como sea necesario para todos los controladores. En la figura 3.1a se muestra de forma conceptual la creación de los hilos en un ejemplo con 2 controladores.

El problema con esta aproximación es que impone la restricción de que todos los controladores se creen al principio del bloque, lo cual es un problema.

```
void Ctrl_HostTask_Consume(){
    bool finish = false;
    while (!finish) {
        Ctrl_Task *p_task = NULL;
        p_task = Ctrl_TaskQueue_Pop(p_ctrl_host_stream);
        switch(p_task->type){
            ...
            //case for thread spawn
            case CTRL_TASK_TYPE_SPAWNTHREADS:
                #pragma omp parallel num_threads()
                {
                    if(omp_get_thread_num()==0){
                    //thread 0 calls recursively to keep executing host tasks
                        Ctrl_HostTask_Consume();
                    }else{
                        //distribute newly created threads
                    }
                }
                finish=true;
            ...
        }
    }
}
```

Listing 3.1: Extracto de la función para extraer y ejecutar tareas de la cola de host con el detalle de la tarea para crear hilos.

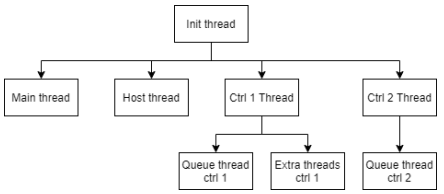
3.2.4. Cuarta opción: el hilo de host crea todos los hilos de forma dinámica

En esta aproximación los hilos extra se crearían cuando el hilo de host ejecute una tarea de un tipo especial. En la ejecución de esta tarea, el hilo de host haría un parallel con el numero de hilos que se desean crear mas uno (el hilo de host debe seguir existiendo). Dentro del pragma parallel el hilo con índice 0 (índice de este nivel de anidamiento) volvería a llamar de forma recursiva a la función que realiza la extracción y ejecución de las tareas de la cola de host. En la figura 3.1 se muestra la parte relevante de esta aproximación. En la figura 3.1a se muestra de forma conceptual la creación de los hilos en un ejemplo con 2 controladores.

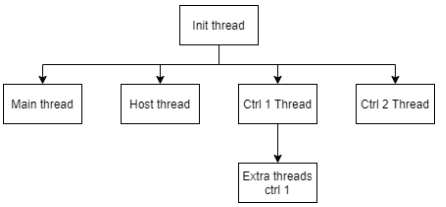
Para aquellos casos en que se prefiera que el hilo de host no tenga estas responsabilidades se podría usar otro hilo dedicado para esto.

3.3. Mecanismos de lanzamiento y sincronización de tareas asíncronas

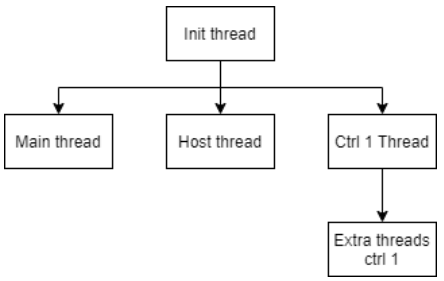
Esta sección describe mecanismos para lanzar las tareas de host y kernels para que se ejecuten de forma asíncrona en sus respectivos núcleos de CPU de forma que se solapen de forma automáticamente cuando no existan dependencias entre ellas.



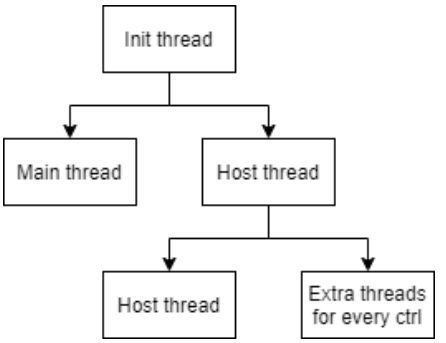
(a) Propuesta 1: Un hilo por ctrl que luego se bifurca si es necesario.



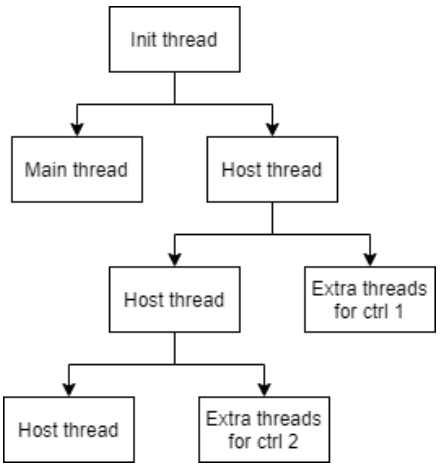
(b) Propuesta 1: Un hilo por ctrl que luego se bifurca si es necesario una vez que se elimine el gestor de cola.



(c) Propuesta 2: Un hilo por ctrl que lo necesita que luego se bifurca si es necesario.



(d) Propuesta 3: el hilo de host task se bifurca para formar todos los hilos de todos los controles al principio del programa.



(e) Propuesta 4: el hilo de host crea todos los hilos de forma dinámica.

Figura 3.1: Diagramas que muestran la creación de los hilos de Controllers según las diferentes propuestas. Las cajas representan los hilos en el programa y las flechas indican que se inicia una región paralela.

Para resolver este problema se plantearon varias soluciones:

3.3.1. Primera aproximación: Tareas de OpenMP con dependencias

Los kernels y tareas de host se lanzan como tareas asíncronas de OpenMP marcando las dependencias con la clausula *"depend"*. Esto tiene varios problemas:

- **Afinidad de los hilos**: una limitación importante de las tareas asíncronas de OpenMP es que no se puede decidir en que hilo se ejecutan, simplemente eligen un hilo libre y empiezan a ejecutarse, esto es un problema para nosotros porque nos impide elegir que núcleo o grupo de núcleos ejecutarán cada cosa.
- **Dependencias de las tareas de OpenMP**: el sistema de dependencias entre tareas que ofrece OpenMP tampoco era óptimo porque debido a la forma en la que funciona internamente (creando un grafo completo con todas las tareas) introduce mucho sobrecoste.

3.3.2. Segunda aproximación: Ejecución en hilo específico y eventos de OpenCL

El problema principal con la versión anterior era no poder decidir que hilo ejecuta las tasks de OpenMP. Para solucionar este problema ideamos un sistema de sincronización en el cual tenemos un hilo concreto esperando para ejecutar una función dada en un puntero global con unos argumentos dados también por punteros globales (en adelante llamado hilo ejecutor) y otro hilo que pone la función y argumentos correspondientes en los punteros globales (en adelante llamado hilo wrapper). La figura 3.2 muestra el funcionamiento de este sistema.

Las funciones que ejecutarían los hilos wrappers se lanzan con tasks de OpenMP (la afinidad de estos hilos no es muy importante ya que estas funciones no llevan carga computacional), pondrían la función y argumentos en los correspondientes punteros globales, daría paso al hilo ejecutor correspondiente y esperaría hasta que el hilo ejecutor haya terminado la función.

Para la sincronización de las tareas solo hace falta sincronizar los wrappers ya que esperan a que el ejecutor termine la tarea. Por tanto se podría hacer de la misma forma que en la aproximación anterior, pero vistos los problemas de las dependencias entre tareas de OpenMP, decidimos usar los eventos de usuario de OpenCL. Estos eventos funcionan bien, pero en conjunto esta solución tiene múltiples problemas:

- **Requiere hilos extra**: esta solución requiere hilos extra para los wrappers, los otros hilos ya existentes en el programa no pueden ejecutar los wrappers lanzados con las tasks de OpenMP porque están ocupados ejecutando otras cosas. Así que haría falta un hilo wrapper para cada hilo ejecutor.

```
// Comm. params
void (*task_name)(int) = NULL;
void *params = NULL;
omp_lock_t lockR;
omp_lock_t lockW;

// kernel function
void my_task(int a) {
    //do task
}

// Wrap function to invoke for a
    kernel
void wrap_my_task(void *a) {
  #pragma omp critical(C2)
  omp_set_lock( &lockW );

  params = a;
  task_name = my_task;

  omp_unset_lock( &lockW );
  omp_unset_lock( &lockR );
  #pragma omp critical(C1)
  omp_set_lock( &lockR );
}
```

```
// Queue manager code
...
    #pragma omp task
    {
      wrap_my_task(kernel_params)
    }
...
```

```
// Task executor code
...
    #pragma omp critical(C1)
    {
        omp_unset_lock( &lockW );
        omp_set_lock( &lockR );
    }

    task_name( *(int *)params );
    task_name = NULL;
    params = NULL;

    #pragma omp critical(C2)
    {
        omp_unset_lock( &lockR );
        omp_set_lock( &lockW );
    }
...
```

Figura 3.2: Pseudocódigo del sistema para pasar el control de la ejecución a otro hilo.

- **No es conforme con el concepto de Propiedad de las locks de OpenMP**: el modelo de sincronización propuesto resultó no ser correcto al no respetar que las locks de OpenMP son propiedad de la región de tarea que bloquea la lock y una lock solo puede ser desbloqueada por una región de tarea que tenga la propiedad de la lock.
- **Introduce una dependencia con OpenCL**: para poder usar los eventos de usuario de OpenCL, hace falta crear un contexto de OpenCL con la plataforma y el dispositivo utilizado, cosas que el usuario debería proveer en la creación del ctrl. No queríamos tener esta dependencia ni que el usuario tenga que preocuparse de estos asuntos.

3.3.3. Aproximación final: Implementar sistema de colas y eventos

En esta aproximación creamos nuestro propio sistema de colas y eventos tratando de reproducir la funcionalidad de los modelos de streams y eventos de CUDA/OpenCL utilizando OpenMP, también creamos una interfaz para los eventos de las tecnologías de otros backends con la idea de que el lanzamiento de tareas de host se realice mediante uno de estos "streams de CPU" creados por nosotros.

Streams de CPU

Para su implementación reutilizamos las colas originales del controlador. Ampliándolas para aceptar tareas de ejecución y un nuevo tipo de tarea para esperas a eventos. Se añade también a la cola un contador para el número de la última tarea finalizada, que se actualiza o lee atómicamente. La cola debe tener una funcionalidad para leer atómicamente y devolver el número de la última tarea finalizada, ya que si usamos el número de la tarea en la cabeza de la cola, la única opción segura es devolver la cabeza-2, ya que la tarea anterior se puede seguir ejecutando o ya haber finalizado. De esta forma, las tareas de host y los kernels de CPU deben tener cada uno su cola independiente y cada cola tener un hilo ejecutor dedicado.

Modelo de eventos de CUDA

El modelo de eventos de CUDA tiene las siguientes propiedades y funcionalidades que trataremos de replicar usando OpenMP:

- Las funciones siempre se llaman desde el codigo del host.
- Crear evento: se inicializa con el estado activado.
- Grabar evento en un stream: el evento registra el stream y el conjunto de trabajo en la cola de ese stream.
- Cada vez que se graba un evento, se olvida la información previa sobre el stream y el conjunto de trabajo.

- Esperar por evento: Bloquea el código de host hasta que finaliza el conjunto de trabajo registrado en el evento en ese momento.
- Esperar por evento en stream: Realiza una copia del estado registrado en el evento. Se pone en el stream una tarea de espera. Cuando se ejecuta la tarea de espera, bloquea el stream hasta que el conjunto de trabajo especificado en la copia del evento (normalmente en otro stream) ha terminado.

Eventos de CPU y eventos genéricos

Reproducción de los eventos de CUDA en OpenMP:

- Crear evento: Inicializar la estructura con los valores iniciales stream=NULL y tarea=0.
- Grabar evento: actualizar el evento: (1) el puntero del stream; (2) Obtener del stream el numero de la ultima tarea introducida.
- Esperar por evento: Bloquea la ejecución mediante sondeo: Un bucle leyendo un valor hasta que una condición se cumpla. El valor leído, con una operación de lectura atómica, es el número de última tarea terminada. La condición es que el número de la última tarea finalizada sea igual o superior que el número de tarea registrado en el evento.
- Esperar por evento en stream (para eventos genéricos) : Hacer una copia del evento con su estado actual. Introducir en el stream una tarea de espera a evento con esa copia. Cuando la cola encuentra una tarea de espera a evento, hace una espera por el evento (como se describió en el anterior punto) con la copia del evento encontrada en la tarea.
- Eventos de usuario: Estructura diferente que consiste en un puntero a un booleano que indica su estado, función de espera diferente (esperar hasta que el estado sea verdadero) y no se permite llamar a grabar evento. Se introduce una función para marcar el evento como *completado* (señalar evento).
- Señalar evento: Cambia el estado de un evento de usuario de CPU a *Completado*.
- Señalar evento en stream (para eventos genéricos): Encolar en el stream una tarea de señal para un evento genérico, solo permitido para eventos de usuario de CPU y eventos de usuario de OpenCL.

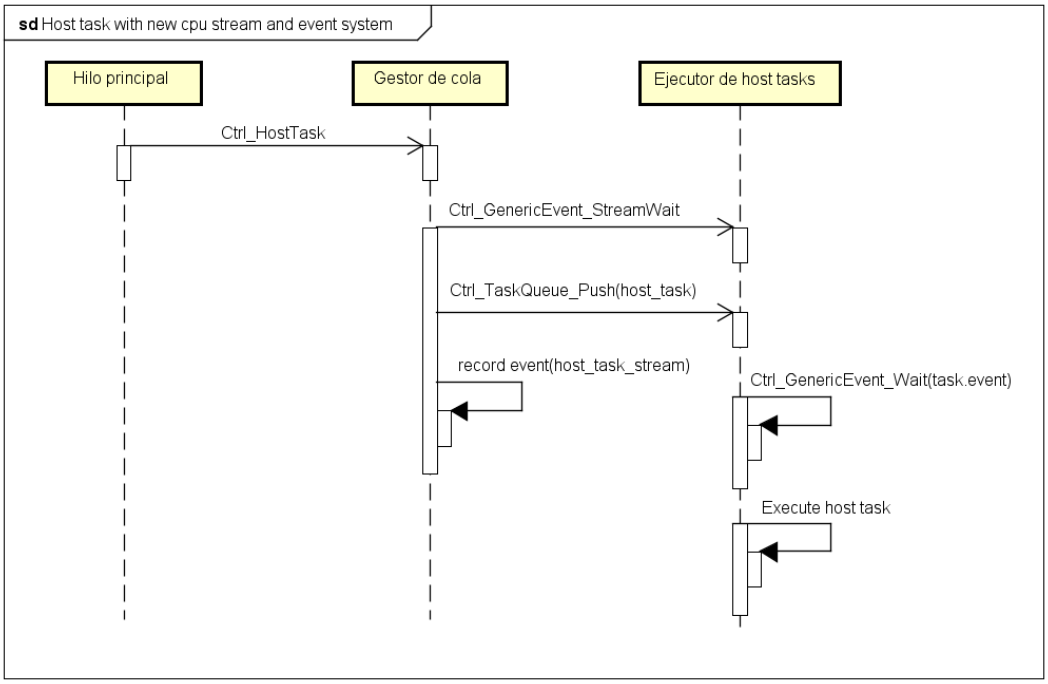


Figura 3.3: Secuencia del lanzamiento de una tarea de host con streams de CPU. Las operaciones Ctrl_GenericEvent_StreamWait, Ctrl_GenericEvent_Wait y record_event representan los eventos correspondientes a cada los parámetro de la tarea de host dependiendo de sus roles.

Capítulo 4

Implementación de la solución

En este capitulo se describe la implementación realizada sobre el prototipo de Controllers. Esta implementación se ha realizado de forma iterativa, comenzando por implementar una versión base y añadiendo funcionalidades y optimizaciones de forma progresiva.

4.1. Modo síncrono

En una primera aproximación implementamos una versión base del nuevo backend en la que sólo funciona el modo síncrono. En el modo síncrono no existe solapamiento entre las tareas ya que el hilo gestor de cola espera a que haya acabado la tarea anterior antes de extraer la siguiente de la cola. Como en esta primera versión sólo hay una tarea en ejecución al mismo tiempo, el hilo gestor de cola que extrae las tareas enviadas al controlador se ocupa también de su ejecución.

Se implementan las funciones para evaluar las tareas que se extraen de la cola del controlador así como las estructuras de la implementación de CPU del controlador y de las tiles y las funciones para crearlos y destruirlos. En esta versión también se implementan los macros que permiten la paralelización de los kernels mediante OpenMP consiguiendo así una granularidad apropiada para un dispositivo de CPU. En el listing 4.1 se muestra un fragmento del macro en el que introducimos los bucles que generan la paralelización con OpenMP.

En esta primera versión no existen transferencias entre host y acelerador. Por lo tanto las operaciones de Controllers para hacer movimientos de memoria estarán vacías y no hacen nada. De forma similar las operaciones de espera simplemente son un punto de sincronización entre el hilo principal y el hilo gestor de cola ya que todas las tareas son secuenciales y se ejecutan de forma síncrona.

Las tareas de host son simplemente una llamada a una función con el código del usuario dentro y los kernels son una llamada a una función con 1, 2 o 3 bucles (dependiendo del

```
#define CTRL_KERNEL_WRAP_CPU( name, argsList, type, ... ) \
{ \
...
    /* 3D thread block */
    _Pragma("omp parallel for num_threads(request.cpu.n_cores)")  \
    for (int i_aux = 0; i_aux < threads.x; i_aux++) { \
        for (int j_aux = 0 ; j_aux < threads.y; j_aux++) { \
            for( int k_aux = 0; k_aux < threads.z; k_aux++) { \
                Ctrl_Thread thread_id; \
                thread_id.x = i_aux; \
                thread_id.y = j_aux; \
                thread_id.z = k_aux; \
                /* Call kernel function defined by user */ \
                Ctrl_Kernel_Cpu_##type##_##name( thread_id,
    CTRL_KERNEL_ARG_LIST_ACCESS_KTILE( argsList, __VA_ARGS__ ) ); \
            } \
        } \
    } \
...
```

Listing 4.1: Fragmento del macro CTRL_KERNEL_WRAP_CPU para un kernel de 3 dimensiones.

número de dimensiones) que simulan el bloque de hilos que se usaría en CUDA u OpenCL, en el interior de estos bucles se llama a una función que contiene el código definido por el usuario. La paralelización se realiza sobre el mas externo de estos bucles mediante la directiva parallel for de OpenMP con el numero de hilos especificados en la creación del ctrl.

4.2. Modo asíncrono

En la segunda etapa implementamos el modo asíncrono para el backend de CPU. En el modo asíncrono se produce solapamiento entre las tareas si sus dependencias lo permiten. Por lo tanto, las tareas no deben ser ejecutadas por el gestor de cola sino lanzadas de forma asíncrona para que otro hilo se ocupe de su ejecución teniendo en cuenta las dependencias de la tarea en concreto. De esta forma el gestor de cola puede continuar extrayendo y lanzando las siguientes tareas permitiendo así su solapamiento.

4.2.1. Creación de hilos y gestión de afinidades

El primer reto de hacer funcionar el modo asíncrono es la creación del hilo que ejecutará los kernels y la gestión de la afinidad de todos los hilos de Controllers.

Las diferentes opciones propuestas para adaptar la creación de los hilos de Controllers se discutieron de forma conceptual en la sección 3.2. En la opción elegida se añade un parámetro al macro `__Ctrl_block__` que representa el número de controladores que necesitan hilos extra que se usarán en el programa. En la figura 4.1 se muestra el código de las funciones para crear y distribuir los hilos que se crean al principio del programa.

```
#define __ctrl_block__(num_ctrl, num_ctrl_threads)        \
  omp_set_nested(1);                                      \
  omp_set_num_threads(2 + num_ctrl_threads);          \
  _Pragma("omp parallel") if (Ctrl_Thread_init() == 0)

```

Listing 4.2: Macro Ctrl_block. Se ocupa de crear los hilos al principio del programa.

```
int Ctrl_Thread_init() {
    #pragma omp single
    {
        hwloc_topology_init(&topo);
        hwloc_topology_load(topo);
    }
    hwloc_obj_t obj;
    if (omp_get_thread_num() == 0) { //main thread
        // bind thread to first PU
        obj = hwloc_get_obj_by_type(topo, HWLOC_OBJ_PU, 0);
        if (obj){
            hwloc_set_cpubind(topo, obj->cpuset, HWLOC_CPUBIND_THREAD);
        }
        return 0; //main thread returns 0
    } else if (omp_get_thread_num() == 1) { //host task executor
        // bind thread to first PU of core 1
        obj = hwloc_get_obj_by_type(topo, HWLOC_OBJ_CORE, 1);
        if (obj){
            hwloc_set_cpubind(topo, obj->first_child->cpuset,
    HWLOC_CPUBIND_THREAD);
        }
        Ctrl_Thread_HostTask_Thread();
        #pragma omp barrier
        #pragma omp barrier
    } else if (omp_get_thread_num() == 2){     //spawner thread
        Ctrl_Thread_EvalThread(topo);
    }
    //destroy hwloc topology
    #pragma omp single nowait
    {
        hwloc_topology_destroy(topo);
    }
    return 1; // other threads return 1 at the end of the program
}

```

Listing 4.3: Función Ctrl_thread_init que se ocupa de distribuir y fijar la afinidad de los hilos que se crean en el ctrl_block.

Figura 4.1: Creación de los hilos iniciales en Controllers

Los hilos extra, en caso de ser necesarios, se crean en la función `Ctrl_Thread_EvalThread`. Para distribuir los hilos extra a sus respectivos cometidos, se introduce en el backend de CPU la función `Ctrl_Cpu_ThreadInit`. Todos los backends que se añadan en el futuro que requieran hilos extra deberán tener una función similar. En la figura 4.2 se muestran estas funciones.

Junto con la implementación del nuevo sistema de creación de hilos de Controllers también es necesario fijar la afinidad de estos hilos que se crean. Esto es necesario para que el hilo de kernels se ejecute en los nodos NUMA indicados por el programador en la creación del controlador y para evitar que los otros hilos de Controllers salten de un núcleo a otro o compitan por los mismos recursos reduciendo el rendimiento. La gestión de afinidades se realiza mediante la librería hwloc que se explica en la sección 2.3.

Determinamos una política de afinidades para decidir a que unidad de procesamiento corresponde cada hilo de Controllers:

- Hilo principal: unidad de procesamiento 0 del core 0.
- Hilo de host: unidad de procesamiento 0 del core 1.
- Hilo gestor de cola del controlador n: unidad de procesamiento 0 del core n+2.
- Hilo de ejecución de kernels en CPU: nodos NUMA especificados por el usuario en la creación del controlador.

El motivo de está política concreta es permitir que cada uno de los hilos de gestión de Controllers tenga su propio núcleo de CPU para maximizar el rendimiento y evitar que dos de esos hilos se "peleen" por tiempo de ejecución en el mismo núcleo. La gestión de estas afinidades se puede observar en los códigos de las figuras 4.1 y 4.2.

4.2.2. Sistema de colas y eventos

El segundo reto del modo asíncrono es lanzar tareas de forma asíncrona a los hilos permitiendo la sincronización de estas tareas acorde con sus dependencias. Las diferentes propuestas para resolver este problema se discuten de forma conceptual en la sección 3.3. En esta sección ahondaremos en las estructuras usadas para implementar está solución.

Streams de CPU: los streams de CPU se han implementado reutilizando las estructuras usadas para las colas de tareas que ya existían en Controllers `Ctrl_TaskQueue` y `Ctrl_Task`. Ampliándolas para aceptar tareas de ejecución y un nuevo tipo de tarea para esperas a eventos. Se añade también a la cola un contador para el número de la última tarea finalizada, que se actualiza o lee atómicamente. Estas estructuras se muestran en la figura 4.3.

```
void Ctrl_Thread_EvalThread() {
    ... //wait for ctrl to be created
    #pragma omp parallel num_threads(Ctrl_GetNumThreads(&ctrl))
    {
        if(omp_get_thread_num()==0){    //queue manager
            // bind thread to first PU of core 2
            hwloc_obj_t obj = hwloc_get_obj_below_by_type(topo,
    HWLOC_OBJ_NUMANODE, host_node, HWLOC_OBJ_CORE, 2);
            if (obj){
                hwloc_set_cpubind(topo, obj->first_child->cpuset,
    HWLOC_CPUBIND_THREAD);
            }
            Ctrl_ConsumeTaskQueue(&ctrl);
            ... //destroy ctrl
        }else{                          //extra threads
            switch(ctrl.type){
                #ifdef _CTRL_ARCH_CPU_
                    case CTRL_TYPE_CPU:
                        Ctrl_Cpu_ThreadInit(&(ctrl.p_impl->cpu), topo,
    host_node);
                        break;
                #endif // _CTRL_ARCH_CPU_
            }
        }
    }
    ...
}

```

Listing 4.4: Función Ctrl_Thread_EvalThread. Se ocupa de crear los hilos extra de cada ctrl.

```
void Ctrl_Cpu_ThreadInit(Ctrl_Cpu *p_ctrl, hwloc_topology_t topo, int node){
  if(omp_get_thread_num()==1){    //kernel thread
      //set affinity
    if (!hwloc_bitmap_iszero(p_ctrl->device_cpuset)){
      hwloc_set_cpubind(topo, p_ctrl->device_cpuset, HWLOC_CPUBIND_THREAD);
    }
    //send thread to do it's thing
    Ctrl_Cpu_StreamConsume(p_ctrl->p_kernel_stream);
  }
}

```

Listing 4.5: Función Ctrl_Cpu_threadInit que se ocupa de distribuir y fijar la afinidad de los hilos extra de un ctrl.

Figura 4.2: Creación de los hilos extra de un ctrl de CPU

```
typedef struct Ctrl_TaskQueue{
    int read;                   /**< Index of next task to execute */
    int write;                  /**< Index of next free spot */
    int last_finished;          /**< Index of last finished task */
    Ctrl_Task buffer[50000];    /**< Buffer of tasks */
} Ctrl_TaskQueue;
```

Listing 4.6: Estructura Ctrl_TaskQueue que representa una cola de tareas.

```
typedef struct Ctrl_Task {
    int device_id;                      /**< The device id inside the Ctrl */
    Ctrl_TaskType task_type;            /**< A task label (Future optimization
    : Reuse predefined tasks) */
    void (*pfn_kernel_wrapper)
      (
        Ctrl_Request request, Ctrl_ImplType impl_type, int impl,
        Ctrl_Thread threads,
        void *p_arguments
      );                          /**< Kernel launching wrapper pointer */
    void (*pfn_hostTask_wrapper)
            (void *p_arguments);        /**< Host Task launching wrapper
    pointer */
    int n_arguments;                    /**< Number of arguments/roles/
    pointers */
    void *p_arguments;                  /**< Packed list of arguments */
    char *p_roles;                      /**< Input/Output roles, for memory
    optimizations */
    void **pp_pointers;                 /**< Pointers to the original
    variables, for memory basic operations */
    uint16_t *p_displacements;          /**< Displacement of parameter over
    arguments array, for memory basic operations */
    struct Ctrl_Task *p_next;           /**< Next task in the queue */
    Ctrl_Thread threads;                /**< Index domain where the task is
    executed */
    HitTile *p_tile;                    /**< For 1 tile tasks */
    Ctrl_GenericEvent event;            /**< For event related tasks */
    Ctrl_Request request;               /**< Contains specific parameters for
    kernel launch  */
    Ctrl_ImplType impl_type;            /**< Kernel implementation type (
    generic, cpu, cuda...) */
} Ctrl_Task;
```

Listing 4.7: Estructura Ctrl_Task que representa las tareas que se pueden introducir en la cola.

Figura 4.3: Estructuras usadas para los streams de CPU.

Eventos de CPU: los eventos de CPU se plantean para reproducir la funcionalidad de la API de eventos de CUDA/OpenCL usando OpenMP y los streams de CPU mencionados anteriormente. Para esto se introducen dos nuevas estructuras, `Ctrl_CpuEvent` y `Ctrl_CpuUserEvent` que representan eventos de CPU y eventos de usuario de CPU. La diferencia entre estos eventos es que mientras que un evento normal se usa para esperar a una tarea encolada en un stream, un evento de usuario espera hasta que se cambie su estado a

```
typedef struct Ctrl_CpuEvent{
    struct Ctrl_TaskQueue *stream;
    int task;
}Ctrl_CpuEvent;
```

Listing 4.8: Estructura Ctrl_CpuEvent.

```
typedef struct Ctrl_CpuUserEvent{
    bool *state;
}Ctrl_CpuUserEvent;
```

Listing 4.9: Estructura Ctrl_CpuUserEvent.

completado mediante una llamada a una función. Las estructuras usadas se pueden ver en las figuras 4.8 y 4.9

Un evento de CPU almacena un puntero a un stream de CPU y un índice de tarea para identificar a que tarea debe esperar. Estos valores se obtienen mediante la función `Ctrl_CpuEvent_Record` que recibe un puntero a un stream de CPU y usa como índice de tarea el índice de la ultima tarea encolada en ese stream. Para esperar a un evento de CPU se usa la función `Ctrl_CpuEvent_Wait`. Esta función realiza una espera activa comprobando de forma atómica el índice de la ultima tarea completada del stream contenido en el evento hasta que este es mayor o igual que el índice de tarea almacenado en el evento.

Un evento de usuario de CPU almacena un puntero a una flag que marca su estado, 0 para *no completado* 1 para *completado*. Cuando se crea un evento de usuario, mediante la función `Ctrl_Cpu_UserEvent_Create`, esta flag se inicializa al estado *no completado*. Para esperar a un evento de usuario de CPU se usa la función `Ctrl_CpuUserEvent_Wait`. Esta función realiza una espera activa comprobando de forma atómica el estado de la flag del evento. Para cambiar el estado de la flag de un evento de usuario se usa la función `Ctrl_CpuUserEvent_Signal`. Es importante que la flag sea un puntero a un entero o booleano para que al señalar un evento cambie el valor de todas las copias del evento que se hayan introducido en streams.

Eventos genéricos: los eventos genéricos están pensados como una interfaz que permita utilizar los eventos de CUDA, OpenCL y CPU en los streams de CPU. Esto nos permite lanzar las tareas de host usando un stream de CPU en todos los backends de Controllers independientemente de la tecnología que usen. Estos eventos genericos se implementan mediante la estructura `Ctrl_GenericEvent` que se muestra en la figura 4.10. Esta estructura contiene un enum que marca el tipo de evento que contiene y una union que contiene el evento concreto.

Junto con está estructura se añaden funciones que permiten encolar en un stream de CPU una espera a un evento genérico (`Ctrl_GenericEvent_StreamWait`) y encolar una operación de señalar un evento generico en un stream de CPU (`Ctrl_GenericEvent_StreamSignal`). También se añaden funciones que permiten esperar o señalar un evento genérico directamente.

Estos eventos y streams se usan para implementar el lanzamiento de tareas de host en

```
typedef struct Ctrl_GenericEvent{
    Ctrl_EventType event_type;
    union event {
        Ctrl_CpuEvent event_cpu;
        Ctrl_CpuUserEvent user_event_cpu;
        #ifdef _CTRL_ARCH_CUDA_
        cudaEvent_t event_cuda;
        #endif //_CTRL_ARCH_CUDA_

        #ifdef _CTRL_ARCH_OPENCL_GPU_
        cl_event event_cl;
        #endif //_CTRL_ARCH_OPENCL_GPU_
   }event;
} Ctrl_GenericEvent;
```

Listing 4.10: Estructura Ctrl_GenericEvent.

todos los backends y para el lanzamiento de kernels en el backend de CPU. La secuencia de lanzamiento de tareas de host consiste en: (1) Cuándo el gestor de cola recibe una tarea de host comprueba los argumentos de esa tarea y sus roles. (2) En función de los roles de estos argumentos, encola esperas a los eventos correspondientes en el stream de host tasks usando eventos genéricos. (3) Encola la tarea de ejecución de la tarea de host en el stream de host tasks. (4) En base a los roles de los argumentos actualiza los eventos correspondientes con el nuevo estado del stream de host tasks. El código del listing 4.11 muestra como se realiza está función.
La implementación del lanzamiento de los kernels del backend de CPU funciona de forma similar.

Modificaciones en los otros backends

Debido a las limitaciones y diferencias de las APIs de eventos que un evento genérico debe soportar, algunas funcionalidades de los eventos genéricos no son posibles de implementar para todos los tipos de eventos. Por lo tanto implementar las tareas de host para los backends de CUDA y OpenCL utilizando streams de CPU y eventos genéricos ha requerido algunas modificaciones.

OpenCL: los eventos de OpenCL no tienen una función *record* para actualizar la información contenida en ellos. Por lo tanto, la forma de trabajar con ellos es crear un nuevo evento para cadaoperación que requiera sincronización y gestionar la vida de estos eventos mediante las funciones `clRetainEvent` y `clReleaseEvent` que incrementan y decrementan el contador de referencias respectivamente. Cuando este contador llega a 0 el evento se elimina.
En nuestra implementación lo que hacemos es llamar a `clRetainEvent` de todos los eventos a los que deseamos esperar antes de lanzar la tarea de host (para que no se elimine mientras estamos esperando por él), y llamar a `clReleaseEvent` de todos los eventos a los que hayamos llamado a `clRetainEvent` y a los que queramos reemplazar con nueva información una vez que la ejecución de la tarea de host haya terminado.
El problema está en las llamadas a `clReleaseEvent` ya que se deben realizar después de

```
void Ctrl_Cpu_EvalTaskHostTaskLaunch(Ctrl_Cpu *p_ctrl, Ctrl_Task *p_task) {
  Ctrl_GenericEvent host_task_event = CTRL_GENERIC_EVENT_NULL;
  host_task_event.event_type = CTRL_EVENT_TYPE_CPU;
    //send wait events depending on role params and transfer mode
  for (int i = 0; i < p_task->n_arguments; i++) {
    if (p_task->p_roles[i] != KERNEL_INVAL) {
      HitTile *p_tile = (HitTile *)(p_task->pp_pointers[i]);
      Ctrl_Cpu_Tile *p_tile_data = (Ctrl_Cpu_Tile *)(p_tile->ext);

      if(p_ctrl->mem_moves){
        if (p_task->p_roles[i] != KERNEL_OUT) {
          if (p_tile_data->last_update == CTRL_TILE_LAST_UPDATE_DEV) {
            Ctrl_Cpu_EvalTaskMoveFromInner(p_ctrl, p_tile);
          }
        }
        host_task_event.event.event_cpu = p_tile_data->
    offloading_last_read_event;
        Ctrl_GenericEvent_StreamWait(host_task_event, p_ctrl_host_stream);
      }

      host_task_event.event.event_cpu = p_tile_data->host_last_write_event;
      Ctrl_GenericEvent_StreamWait(host_task_event, p_ctrl_host_stream);

      if (p_task->p_roles[i] != KERNEL_IN) {
        if(p_ctrl->mem_moves){
          p_tile_data->last_update = CTRL_TILE_LAST_UPDATE_HOST;
          host_task_event.event.event_cpu = p_tile_data->
    offloading_last_write_event;
          Ctrl_GenericEvent_StreamWait(host_task_event, p_ctrl_host_stream);
        }
        host_task_event.event.event_cpu = p_tile_data->host_last_read_event;
        Ctrl_GenericEvent_StreamWait(host_task_event, p_ctrl_host_stream);
      }
    }
  }
...
    //launch host task
  Ctrl_TaskQueue_Push(p_ctrl_host_stream,*p_task);
    //record events depending on roles
  for (int i = 0; i < p_task->n_arguments; i++) {
    if (p_task->p_roles[i] != KERNEL_INVAL) {
      HitTile *p_tile = (HitTile *)(p_task->pp_pointers[i]);
      Ctrl_Cpu_Tile *p_tile_data = (Ctrl_Cpu_Tile *)(p_tile->ext);
      if (p_task->p_roles[i] != KERNEL_IN) {
        Ctrl_CpuEvent_Record(&p_tile_data->host_last_write_event,
    p_ctrl_host_stream);
      }

      if (p_task->p_roles[i] != KERNEL_OUT) {
        Ctrl_CpuEvent_Record(&p_tile_data->host_last_read_event,
    p_ctrl_host_stream);
      }
    }
  }
...
}
```

Listing 4.11: Código del lanzamiento de tareas de host en CPU con el nuevo sistema.

que la tarea de host haya terminado de ejecutarse, lo que implica que debe llamarse desde el stream. Para solucionar esto añadimos la función `Ctrl_GenericEvent_StreamRelease` que encola en un stream de CPU una operación de *release*. Esta función, como se puede suponer, únicamente funciona con eventos genéricos que contienen un evento de OpenCL.

CUDA: los eventos de cuda tienen 2 problemas:

- No soportan eventos de usuario ni ninguna forma de señalar eventos desde el host, por lo tanto, no se puede señalar los eventos cuando la tarea de host termina.
- No se pueden realizar copias de los eventos, lo cual es necesario para que cuando se hace *record* de un evento que se ha introducido en un stream previamente, el evento que está en el stream no sea modificado.

La solución para estos problemas es usar conjuntamente los streams de CPU y la función de callback asíncrono de CUDA `cudaLaunchHostFunc`. Esta función es ejecutada por el hilo del driver CUDA asociado al dispositivo. Este hilo se ejecuta en la máquina host y es independiente de la arquitectura de Controllers. Una función de callback debe ser asignada a un stream de cuda como una operación mas. Esta función será ejecutada cuando a dicha operación le llegue el turno de ejecución en el stream de CUDA. El hilo del driver tiene una cola de callbacks pendientes a la que se irán incluyendo aquellas que estén preparadas para ser ejecutadas; este hilo las evaluará de forma sincrónica mediante política FIFO.

Para solucionar nuestros problemas realizamos la espera a los eventos en un stream de CUDA al cual posteriormente lanzamos una `cudaLaunchHostFunc` cuya función de callback (`Ctrl_Cuda_LaunchHost`) llamará a una función que a su vez lanzará al stream de CPU la tarea de host y esperará a que esta termine de ejecutarse. Después de encolar el *callback* en el stream de CUDA llamamos a *record* de los eventos correspondientes. De esta forma no es necesario encolar esperas a eventos de CUDA en streams de CPU ni señalar eventos de CUDA desde el host. En el listing 4.12 muestra el funcionamiento del nuevo sistema.

4.3. Modo con transferencias de memoria

Una vez que el modo asíncrono está implementado, se da la opción de activar el modo de transferencias de memoria para aprovecharse de la localidad y reducir la latencia en los accesos a memoria desde los kernels.

La forma de realizar estas transferencias es utilizar dos hilos extra (uno para las transferencias desde el host al dispositivo y otro para las del dispositivo al host) cada uno con su stream. Estos hilos usan la función `memcpy()` para copiar los datos desde la estructura de datos del host a la memoria reservada en el nodo NUMA del dispositivo.
De esta forma podemos realizar copias de memoria de forma asíncrona en ambos sentidos.

```
void Ctrl_Cuda_LaunchHost(void* p_task){
    //push host task to host queue
  Ctrl_TaskQueue_Push(p_ctrl_host_stream, *(Ctrl_Task*)p_task);
  //wait until it ends
  Ctrl_CpuEvent e=Ctrl_CpuEvent_Create();
  Ctrl_CpuEvent_Record(&e, p_ctrl_host_stream);
  Ctrl_CpuEvent_Wait(e);
}

void Ctrl_Cuda_EvalTaskHostTaskLaunch(Ctrl_Cuda *p_ctrl, Ctrl_Task *p_task) {
  for (int i = 0; i < p_task->n_arguments; i++) {
    if (p_task->p_roles[i] != KERNEL_INVAL) {
      HitTile *p_tile = (HitTile *)(p_task->pp_pointers[i]);
      Ctrl_Cuda_Tile *p_tile_data = (Ctrl_Cuda_Tile *)(p_tile->ext);
            //move tiles if necessary
      if (p_task->p_roles[i] != KERNEL_OUT) {
        if (p_tile_data->last_update == CTRL_TILE_LAST_UPDATE_DEV) {
          Ctrl_Cuda_EvalTaskMoveFromInner(p_ctrl, p_tile);
        }
      }
            //send wait events depending on param role
      CUDA_OP( cudaStreamWaitEvent(p_ctrl->stream_host, p_tile_data->
    offloading_last_read_event, 0) );
      CUDA_OP( cudaStreamWaitEvent(p_ctrl->stream_host, p_tile_data->
    host_last_write_event, 0) );

      if (p_task->p_roles[i] != KERNEL_IN) {
        p_tile_data->last_update = CTRL_TILE_LAST_UPDATE_HOST;
        CUDA_OP( cudaStreamWaitEvent(p_ctrl->stream_host, p_tile_data->
    offloading_last_write_event, 0) );
        CUDA_OP( cudaStreamWaitEvent(p_ctrl->stream_host, p_tile_data->
    host_last_read_event, 0) );
      }
    }
  }
...
    //launch hostfunc
  CUDA_OP( cudaLaunchHostFunc(p_ctrl->stream_host, Ctrl_Cuda_LaunchHost ,
    p_task) );
    //record events depending on roles
  for (int i = 0; i < p_task->n_arguments; i++) {
    if (p_task->p_roles[i] != KERNEL_INVAL) {
      HitTile *p_tile = (HitTile *)(p_task->pp_pointers[i]);
      Ctrl_Cuda_Tile *p_tile_data = (Ctrl_Cuda_Tile *)(p_tile->ext);
      if (p_task->p_roles[i] != KERNEL_IN) {
        CUDA_OP( cudaEventRecord(p_tile_data->host_last_write_event, p_ctrl->
    stream_host) );
      }
      if (p_task->p_roles[i] != KERNEL_OUT) {
        CUDA_OP( cudaEventRecord(p_tile_data->host_last_read_event, p_ctrl->
    stream_host) );
      }
    }
  }
...
}
```

Listing 4.12: Código del lanzamiento de tareas de host en CUDA con el nuevo sistema.

```
#define CTRL_KERNEL_WRAP_CPU( name, argsList, type, ... ) \
{ \
...
    _Pragma("omp parallel for num_threads(request.cpu.n_cores)")  \
    for (int i_outer = 0; i_outer < threads.x; i_outer+=BLOCKSIZE) { \
        for (int j_outer = 0; j_outer < threads.y; j_outer+=BLOCKSIZE) { \
            for (int k_outer = 0; k_outer < threads.z; k_outer+=BLOCKSIZE) { \
                int i_max = ((i_outer+BLOCKSIZE) > threads.x) ? threads.x : (
    i_outer+BLOCKSIZE); \
                int j_max = ((j_outer+BLOCKSIZE) > threads.y) ? threads.y : (
    j_outer+BLOCKSIZE); \
                int k_max = ((k_outer+BLOCKSIZE) > threads.z) ? threads.z : (
    k_outer+BLOCKSIZE); \
                for (int i_inner = i_outer; i_inner < i_max; i_inner++){ \
                    for (int j_inner = j_outer; j_inner < j_max; j_inner++){ \
                        for (int k_inner = k_outer; k_inner < k_max; k_inner
    ++){ \
                            Ctrl_Thread thread_id = CTRL_THREAD_NULL; \
                            thread_id.x = i_inner; \
                            thread_id.y = j_inner; \
                            thread_id.z = k_inner; \
                            Ctrl_Kernel_Cpu_##type##_##name( thread_id,
    CTRL_KERNEL_ARG_LIST_ACCESS_KTILE( argsList, __VA_ARGS__ ) ); \
                        } \
                    } \
                } \
            } \
        } \
    } \
...
}
```

Listing 4.13: Fragmento del macro CTRL_KERNEL_WRAP_CPU para un kernel de 3 dimensiones con tiling.]Fragmento de codigo para tiling en kernels de CPU.]Fragmento del macro CTRL_KERNEL_WRAP_CPU para un kernel de 3 dimensiones con tiling.

4.4. Tiling en la ejecución de los kernels

Como último paso en la implementación introducimos tiling en la paralelización de los kernels para mejorar el aprovechamiento de las cachés. Esto lo haremos dividiendo los bucles que iteran sobre los el bloque de hilos designado por el programador en bloques (tiles) de tamaño 16 en cada dimensión utilizada. Elegimos 16 como tamaño de bloque porque es uno de los tamaños que suelen funcionar bien para la mayoría de los dispositivos. Esto introduce un problema y es que si el tamaño de la estructura de datos sobre la que trabajamos no es divisible entre el tamaño de bloque tendremos bloques incompletos en los bordes, por lo tanto tendremos que asegurarnos de no salirnos de la matriz. En el listing 4.13 se muestra el código mostrado previamente en el listing 4.1 con las modificaciones necesarias para usar tiling.

Capítulo 5

Experimentación

En este capítulo se introducen los siguientes aspectos:

- Los
- Una descripción de los casos de estudio elegidos.
- Estudio de rendimiento realizado.

5.1. Metodología

5.1.1. Objetivos de la experimentación

El objetivo de esta experimentación es obtener medidas de los tiempos de ejecución de las aplicaciones Matrixpow, Hotspot y Sobel en la máquina Hydra, perteneciente al cluster del grupo Trasgo. A continuación, con estos datos se realiza una comparativa entre los tiempos obtenidos por las versiones de referencia y Controllers.

Con estos resultados podremos validar el modelo propuesto y observar las diferencias de comportamiento entre las diferentes versiones.

5.1.2. Plataforma de ejecución

El estudio experimental se ha llevado a cabo en la maquina *hydra* con las siguientes características:

- Procesador: 2x Intel Xeon E5-2690 v3 @1.9 GHz.

- Nodos NUMA: 2 nodos NUMA con 6 núcleos sin hyperthreading.
- Memoria RAM total: 64GB GDDR3.

El sistema operativo de *hydra* es la distribución de Linux CentOS versión 7.5.1804. Todos los programas se compilan con GCC v7.2.

5.1.3. Escenario de la experimentación

Esta sección expone el estudio de rendimiento realizado que valida el modelo propuesto.

Se han desarrollado versiones síncronas usando la tecnología OpenMP para los casos de estudio escogidos. Nos referiremos a estas como versiones de referencia para las comparativas. La versión de *Controllers* tiene una única implementación ya que la política síncrona o asíncrona y si realizamos transferencias o no se decide en tiempo de ejecución. El estudio de rendimiento comparará los tiempos entre las siguientes versiones:

- Versión de referencia síncrona.
- Versión de Controllers con política síncrona usando copias cero.
- Versión de Controllers con política síncrona usando transferencias de memoria.
- Versión de Controllers con política asíncrona usando copias cero.
- Versión de Controllers con política asíncrona usando transferencias de memoria.

Las versiones de referencia se ejecutan usando 6 hilos. Las versiones de Controllers usando el nodo NUMA 0 como host y el nodo NUMA 1 como device con 6 hilos.

Las ejecuciones de las soluciones desarrolladas se ejecutan variando el tamaño de las matrices de los casos de estudio *hotspot* y *matrix pow*, mientras que en *sobel yuv* se ha probado con 100 fotogramas de un vídeo en alta definición (1920x1080 píxeles). En la tabla 5.1 se muestran el numero de iteraciones y tamaños de matriz usados en los dos primeros casos de uso.

Caso de estudio	Numero de iteraciones	Tamaños de matriz
Hotspot	500	1000, 2500, 5000, 10000, 15000
Matrix pow	40	1024, 1536, 2048, 4092, 6144, 8192

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

Para cada tamaño de matriz y caso de estudio se ejecutan 25 repeticiones. 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)

```
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

5.2. Casos de estudio

Para comprobar el rendimiento de nuestro programa hemos seleccionado tres programas distintos, uno de ellos con 4 variantes, para un total de siete casos de estudio. Estos programas representan diferentes clases de aplicaciones y diferentes escenarios.

- El primero es un programa stencil iterativo de la suite de referencia de Rodinia.
- El segundo es un método iterativo de potencia de matrices.
- El tercero aplica un filtro de imagen Sobel a los fotogramas de un vídeo.

5.2.1. Rodinia: Hotspot stencil computation

El programa base está incluido en la suite de aplicaciones de referencia de Rodinia [5], computa el punto de estabilidad de la Ecuación Diferencial Parcial de Poisson (PDE) de la difusión de calor. Utiliza el método iterativo de Jacobi para un espacio discreto bi-dimensional. Este programa stencil de cuatro puntos ejecuta un número de iteraciones fijo. En cada iteración se calcula un nuevo valor por cada celda de la matriz, utilizando la información de 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.

5.2.2. Matrix pow

Este programa es una evolución de los programas *2mm* y *3mm* de la suite de referencia PolYBench [6] para generar una cadena de multiplicaciones de matrices de longitud arbitraria, computando $C = A^n$ mediante el siguiente método iterativo:

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

```
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

$$C^1 = A$$

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 dividiendo cada elemento por el máximo.
- Calcular la norma de los elementos como la raíz cuadrada de la suma de cada elemento al cuadrado.
- Dividir cada elemento de la matriz por la norma de los elementos.

5.2.3. Sobel YUV

El operador Sobel [7] 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 video 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 imagenes 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:

```
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

- Las imágenes de entrada y salida se leen y escriben directamente de ficheros.
- Las imagenes 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.

5.3. Resultados de la experimentación

A continuación se presentan en gráficas y tablas los resultados obtenidos para Hotspot Matrixpow y Sobel.

5.3.1. Hotspot

Como podemos observar en la gráfica 5.1, las versiones síncronas de Controllers introducen algo de overhead sobre la versión de referencia, siendo este overhead mayor en la versión con transferencias de memoria. También podemos observar que las versiones asíncronas mejoran ambas en rendimiento con respecto a la versión de referencia, siendo un poco más rápida la versión con transferencias de memoria. Esto probablemente se deba a que en la versión asíncrona Controllers es capaz de ocultar el tiempo que se tarda en realizar las transferencias de memoria su solapamiento con la ejecución de kernels y tareas de host. Beneficiándose de los accesos más rápidos a la memoria local del nodo NUMA.

5.3.2. Matrix Pow

Como se puede observar en la gráfica 5.2, las diferencias entre las versiones son muy pequeñas en este caso de estudio, pero si miramos la tabla 5.3, podemos ver que, aunque pequeñas, hay diferencias de tiempo significativas entre las diferentes versiones.
De nuevo observamos que las versiones síncronas de Controllers son más lentas que la versión de referencia. Siendo en este caso muy similares los tiempos entre la versión con transferencias

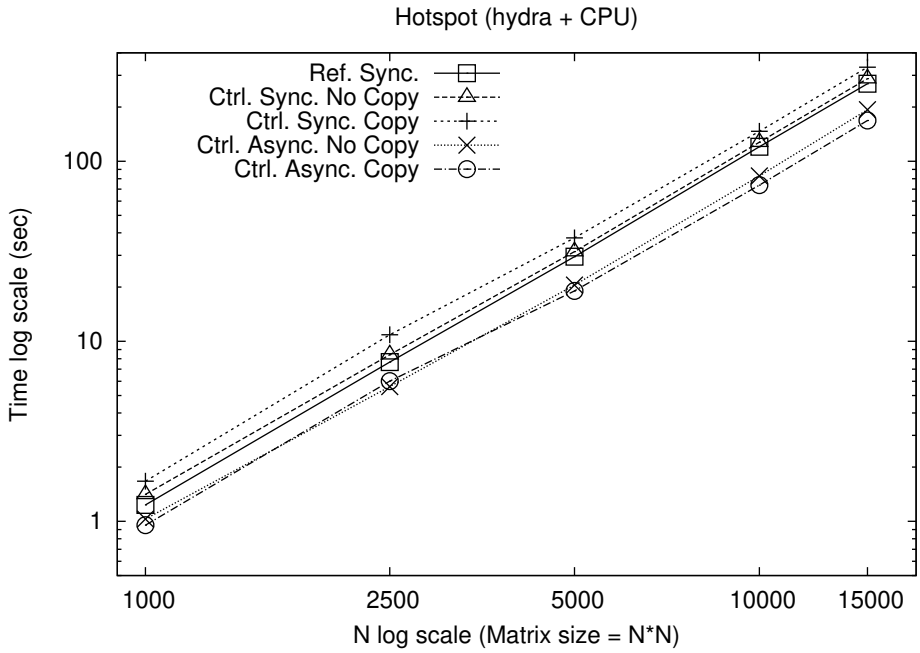


Figura 5.1: Gráfica comparativa de rendimiento - hotspot

y la versión sin ellas. En cuanto a las versiones asíncronas, observamos que salvo en el tamaño más pequeño y en el tamaño más grande, son iguales o un poco más lentas que la versión de referencia, siendo para algunos tamaños prácticamente iguales entre si y para otros ligeramente más rápida la versión con transferencias.

Hotspot (Hydra + CPU)					
size	versión	media	mediana	desviación	IC 95 %
1000	Ref. Sync.	1.2327	1.2329	0.0047	[1.2306, 1.2348]
	Ctrl. Sync. No copy	1.4102	1.4028	0.0206	[1.4017, 1.4187]
	Ctrl. Sync. Copy	1.6712	1.6754	0.0143	[1.6652, 1.6773]
	Ctrl. Async. No copy	1.0289	1.0322	0.0250	[1.0184, 1.0395]
	Ctrl. Async. Copy	0.9514	0.9600	0.0241	[0.9415, 0.9614]
2500	Ref. Sync.	7.6661	7.6950	0.2720	[7.5538, 7.7784]
	Ctrl. Sync. No copy	8.4161	8.2630	0.3149	[8.2861, 8.5461]
	Ctrl. Sync. Copy	13.3435	11.1681	3.1152	[12.0576, 14.6294]
	Ctrl. Async. No copy	5.6020	5.6400	0.2873	[5.4834, 5.7206]
	Ctrl. Async. Copy	8.7424	6.3513	3.1577	[7.4390, 10.0458]
5000	Ref. Sync.	29.4958	29.3751	0.6285	[29.2304, 29.7612]
	Ctrl. Sync. No copy	31.4560	31.2492	0.5285	[31.2379, 31.6741]
	Ctrl. Sync. Copy	37.5405	37.5176	0.0560	[37.5168, 37.5641]
	Ctrl. Async. No copy	20.5364	20.5927	0.3542	[20.3869, 20.6860]
	Ctrl. Async. Copy	19.0449	19.0399	0.0472	[19.0249, 19.0648]
10000	Ref. Sync.	120.4205	120.2926	1.8709	[119.6305, 121.2106]
	Ctrl. Sync. No copy	127.4068	127.4506	2.3575	[126.3873, 128.4262]
	Ctrl. Sync. Copy	146.5913	146.5905	0.0241	[146.5800, 146.6026]
	Ctrl. Async. No copy	82.8784	83.1273	2.2764	[81.9388, 83.8181]
	Ctrl. Async. Copy	73.6941	73.7148	0.1218	[73.6438, 73.7444]
15000	Ref. Sync.	269.1948	269.0740	4.5317	[267.3242, 271.0653]
	Ctrl. Sync. No copy	287.1777	287.3122	6.1653	[284.6328, 289.7227]
	Ctrl. Sync. Copy	332.8508	332.8465	0.0542	[332.8268, 332.8748]
	Ctrl. Async. No copy	193.3253	191.6170	8.7107	[189.7297, 196.9209]
	Ctrl. Async. Copy	168.4115	168.4154	0.1919	[168.3323, 168.4907]

Cuadro 5.2: Datos de la experimentación Hotspot en segundos.

5.3.3. Sobel

Como podemos observar en la tabla 5.4, las versiones síncronas de Controllers introducen algo de overhead, siendo éste algo mayor en la versión sin copias. La versión asíncrona sin copias consigue las mejoras esperadas sobre la versión de referencia y la versión con copias tiene un rendimiento algo menor.

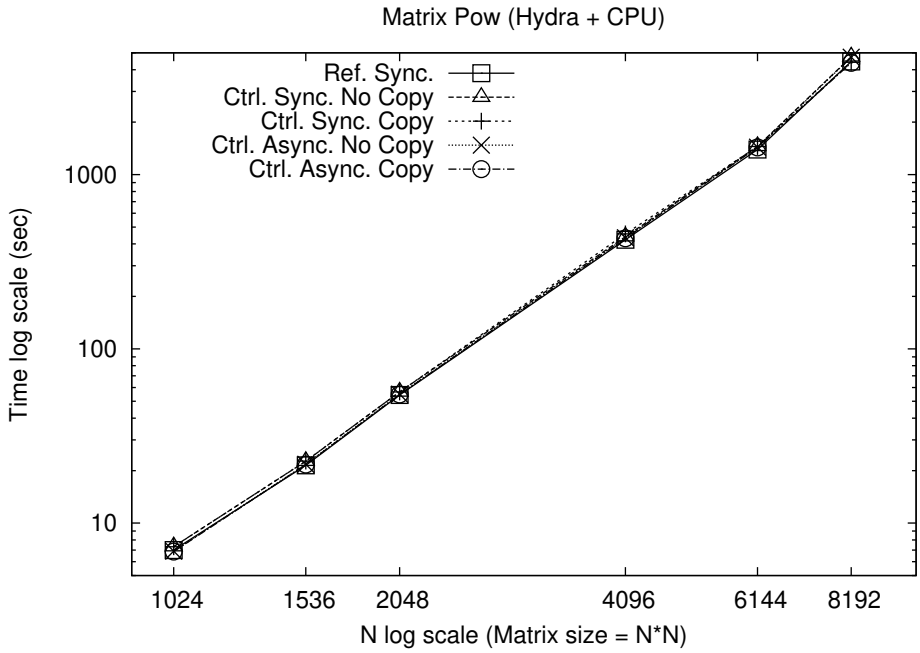


Figura 5.2: Gráfica comparativa de rendimiento - matrix pow

5.4. Conclusiones

En general, los resultados obtenidos muestran que las versiones de Controllers introducen algo de overhead, especialmente visible en las versiones síncronas, pero lo compensa por simplificar la programación de un modelo asíncrono y solapar tareas de forma transparente.

También sabemos que una parte de ese overhead tiene relación con la gestión de colas de los controladores, las cuales se planean eliminar como trabajo futuro.

Matrix Pow (Hydra + CPU)					
size	versión	media	mediana	desviación	IC 95 %
1024	Ref. Sync.	7.0048	7.0050	0.0039	[7.0032, 7.0065]
	Ctrl. Sync. No copy	7.3229	7.3190	0.0194	[7.3149, 7.3309]
	Ctrl. Sync. Copy	7.3156	7.3155	0.0085	[7.3121, 7.3191]
	Ctrl. Async. No copy	6.9069	6.9056	0.0108	[6.9024, 6.9113]
	Ctrl. Async. Copy	6.8609	6.8609	0.0064	[6.8582, 6.8635]
1536	Ref. Sync.	21.4642	21.4617	0.0217	[21.4548, 21.4736]
	Ctrl. Sync. No copy	22.6664	22.6636	0.0125	[22.6611, 22.6717]
	Ctrl. Sync. Copy	22.6830	22.6842	0.0048	[22.6810, 22.6850]
	Ctrl. Async. No copy	21.7466	21.7464	0.0185	[21.7389, 21.7542]
	Ctrl. Async. Copy	21.6658	21.6647	0.0080	[21.6625, 21.6691]
2048	Ref. Sync.	54.3256	54.3241	0.0675	[54.2977, 54.3535]
	Ctrl. Sync. No copy	56.5478	56.5442	0.0183	[56.5401, 56.5556]
	Ctrl. Sync. Copy	56.5795	56.5501	0.0609	[56.5544, 56.6046]
	Ctrl. Async. No copy	54.8815	54.8814	0.0122	[54.8761, 54.8869]
	Ctrl. Async. Copy	54.7455	54.7481	0.0098	[54.7411, 54.7498]
4096	Ref. Sync.	422.7375	422.5047	0.7218	[422.4327, 423.0423]
	Ctrl. Sync. No copy	440.4049	440.4239	0.0817	[440.3704, 440.4394]
	Ctrl. Sync. Copy	457.0341	438.8024	27.9464	[445.4984, 468.5698]
	Ctrl. Async. No copy	433.7472	433.7441	0.0334	[433.7315, 433.7628]
	Ctrl. Async. Copy	431.0379	430.8471	0.4718	[430.8105, 431.2653]
6144	Ref. Sync.	1391.6682	1391.5816	0.5073	[1391.4588, 1391.8777]
	Ctrl. Sync. No copy	1452.7045	1452.7124	0.0917	[1452.6638, 1452.7452]
	Ctrl. Sync. Copy	1449.0341	1449.0180	0.1617	[1448.9642, 1449.1040]
	Ctrl. Async. No copy	1438.1064	1438.0918	0.1089	[1438.0593, 1438.1535]
	Ctrl. Async. Copy	1433.2651	1433.2641	0.1799	[1433.1853, 1433.3448]
8192	Ref. Sync.	4457.4003	4454.3983	11.9619	[4452.4626, 4462.3379]
	Ctrl. Sync. No copy	4741.3905	4742.9226	3.0926	[4740.0531, 4742.7278]
	Ctrl. Sync. Copy	4408.9957	4408.7374	2.5831	[4407.8504, 4410.1409]
	Ctrl. Async. No copy	4715.9555	4716.6061	3.6885	[4714.3202, 4717.5909]
	Ctrl. Async. Copy	4385.1575	4385.2830	2.3116	[4384.1579, 4386.1572]

Cuadro 5.3: Datos de la experimentación Matrix Pow en segundos.

Sobel YUV (Hydra + CPU)					
caso	versión	media	mediana	desviación	IC 95 %
File To File	Ref. Sync.	1.0505	1.0451	0.0104	[1.0462, 1.0547]
	Ctrl. Sync. No copy	1.2487	1.2534	0.0161	[1.2421, 1.2554]
	Ctrl. Sync. Copy	1.1458	1.1443	0.0069	[1.1428, 1.1487]
	Ctrl. Async. No copy	0.8131	0.8128	0.0038	[0.8114, 0.8148]
	Ctrl. Async. Copy	1.1473	1.1463	0.0051	[1.1449, 1.1497]
Mem To File	Ref. Sync.	1.0057	1.0056	0.0015	[1.0050, 1.0063]
	Ctrl. Sync. No copy	1.2721	1.2804	0.0155	[1.2657, 1.2785]
	Ctrl. Sync. Copy	1.1180	1.1198	0.0164	[1.1112, 1.1248]
	Ctrl. Async. No copy	0.8110	0.8119	0.0048	[0.8087, 0.8133]
	Ctrl. Async. Copy	1.1040	1.1053	0.0111	[1.0993, 1.1087]
File To Mem	Ref. Sync.	1.1075	1.1120	0.0082	[1.1042, 1.1109]
	Ctrl. Sync. No copy	1.3088	1.3003	0.0137	[1.3032, 1.3145]
	Ctrl. Sync. Copy	1.2099	1.2079	0.0063	[1.2071, 1.2127]
	Ctrl. Async. No copy	0.8033	0.8098	0.0147	[0.7972, 0.8093]
	Ctrl. Async. Copy	1.1856	1.1874	0.0146	[1.1796, 1.1916]
Mem To Mem	Ref. Sync.	1.0604	1.0607	0.0014	[1.0598, 1.0609]
	Ctrl. Sync. No copy	1.3230	1.3291	0.0157	[1.3165, 1.3294]
	Ctrl. Sync. Copy	1.1844	1.1876	0.0160	[1.1778, 1.1910]
	Ctrl. Async. No copy	0.8326	0.8333	0.0039	[0.8308, 0.8345]
	Ctrl. Async. Copy	1.1691	1.1714	0.0161	[1.1624, 1.1757]

Cuadro 5.4: Datos de la experimentación Sobel YUV en segundos

Capítulo 6

Conclusiones

Este capítulo introduce los siguientes aspectos:

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

6.1. Objetivos cumplidos

Se han alcanzado todos los objetivos propuestos al inicio del proyecto:

- Se ha estudiado las librerías de Hitmap y Controllers, y se ha entendido el modelo previo Controllers.
- Se ha diseñado e implementado el nuevo backend de Controllers para CPUs.
- Se han diseñado e implementado benchmarks para realizar una experimentación sobre el modelo
- Se ha logrado un estudio y evaluación (test de rendimiento) que valida el prototipo del modelo propuesto.

6.2. Trabajo futuro

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

Se ha observado que para algunos tamaños concretos en algunas aplicaciones se pueden producir efectos estocásticos en la ordenación de los eventos que son correctas pero reducen el rendimiento. Estudiar y reducir o eliminar esos efectos.

Este proyecto queda abierto para futuras ampliaciones y modificaciones. La principal linea de continuación de trabajo en Controllers es permitir el uso de varios controladores de igual o distinto tipo a la vez en el mismo bloque de control, permitiendo a su vez conectar *tiles* a varios controladores a la vez de manera que se realicen transferencias de datos entre estos dispositivos a través del *host* de forma automática cuando sea necesario.

Como parte del camino hacia la versión de múltiples controladores una de las opciones que se plantean es la eliminación del hilo gestor de la cola así como de la propia cola de tareas de cada control. En su lugar el hilo principal sería el que lanzaría las tareas al dispositivo conectado al controlador. Esto ayudaría a simplificar la sincronización de los hilos a lo largo de todo el prototipo de *Controllers*.

6.3. Valoración personal

Siendo éste el proyecto más grande del que he formado parte hasta ahora y, como imagino que sucederá en todos los trabajos de fin de grado, ha presentado un gran reto a nivel personal.
Es una fase de aprendizaje donde se acerca el trabajo que se realiza al mundo real, al mundo donde en un futuro próximo formaremos parte de él. Este trabajo de investigación ha tenido una estructura diferente al desarrollo de software convencional, como el que se plantea en las asignaturas del grado. Las etapas en comparación con un desarrollo son más lentas y metodologías ágiles de trabajo pueden no ser convenientes. Opino que el trabajo que he realizado es un buen complemento a lo aprendido en el resto del grado, principalmente por la experiencia práctica. He podido usar lo aprendido en muchas asignaturas, principalmente Computación Paralela pero también muchas otras como las asignaturas de Sistemas Operativos, Fundamentos de Computadoras, o incluso Matemáticas.

Bibliografía

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

[2] 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)*, 32(6):838–853, 2018.

[3] Top500. Top 500 main page. https://www.top500.org. Ultimo acceso Julio de 2020.

[4] V. Lara, I. Taboada, E. Rodriguez-Gutiez, Y. Torres, D. Llanos, and A. Gonzalez-Escribano. Transparent asynchronous data transfers on heterogeneous platforms. 2019.

[5] S. Che and et al. Rodinia: A benchmark suite for heterogenous computing. In Proc. IISWC'09, pages 44–54. IEEE, 2009.

[6] L-N. Pouchet and et al. PolyBench/C, the Polyhedral Benchmark suite, GPU 1.0, 2012. http://web.cs.ucla.edu/~pouchet/software/polybench, Ultimo acceso: Julio, 2020.

[7] R.C. Gonzalez and R.E. Woods. Digital Image Processing (3rd Edition). Prentice Hall, 2007

[8] Difference between GPU and CPU. Nvidia blog. https://blogs.nvidia.com/blog/2009/12/16/whatsthe-difference-between-a-cpu-and-a-gpu/. Ultimo acceso Julio de 2020.

[9] Using OpenMP: portable shared memory parallel programming, Barbara Chapman, Gabriele Jost, Ruud van der Pas. MIT press, 2008, ISBN: 978-0-262-53302-7

[10] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, et al.. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Feb 2010, Pisa, Italy. ff10.1109/PDP.2010.67ff. ffinria-00429889

[11] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. Comm. ACM, 9(3):143–155, 1966.

[12] Grupo Trasgo. Página principal. https://trasgo.infor.uva.es/. Ultimo acceso Agosto de 2020.

[13] P. Sundararajan, "High performance computing using FPGAs," tech. rep., Technical Report. Available Online 2010: www. xilinx. com/support . . . , 2010.

[14] Majo, Zoltan & Gross, Thomas. (2011). Memory System Performance in a NUMA Multicore Multiprocessor. 12. 10.1145/1987816.1987832.

[15] Christopher Hollowell et al 2015 J. Phys.: Conf. Ser. 664 092010.

[16] https://frankdenneman.nl/2016/07/07/numa-deep-dive-part-1-uma-numa/ Ultimo acceso Septiembre de 2020.

Apéndice A

Contenidos del fichero ZIP

- Controllers: contiene el código fuente de la librería Controllers.