Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado

Grado en Ingeniería Informática
(Mención en Computación)

Análisis comparativo de dos modelos de programación paralela heterogénea

---

Alumno:
D. Juan González Caminero

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

Resumen

En la actualidad, es frecuente el uso de coprocesadores de alto rendimiento en sistemas de cómputo de propósito general, tal y como refleja la lista TOP500 de los mayores supercomputadores del mundo. Estos sistemas presentan una serie de desafíos a la hora de desarrollar aplicaciones haciendo uso de estos dispositivos de forma eficiente, uno de ellos es la gestión de memoria. Los coprocesadores tienen su propio espacio de memoria, separado de la memoria de la máquina donde están instalados, y las transferencias de datos entre ambos son costosas.

En este contexto, surgen diversas propuestas para aliviar este problema, algunas de ellas nativas, como las transferencias de memoria asíncronas o la memoria unificada de CUDA, y otras de terceros que en muchos casos pretenden facilitar al programador el uso de estos recursos nativos del modelo de programación.

En este trabajo se realiza una comparativa entre dos de estas soluciones: El modelo Controllers del grupo de investigación Trasgo, y el modelo OmpSs, del Centro de Supercomputación de Barcelona. Aunque no es su única función, ambos ofrecen al desarrollador un sistema automático para introducir transferencias de memoria eficientes.

Para realizar esta comparativa se ha realizado la implementación en OmpSs de las aplicaciones de prueba usadas por el grupo Trasgo en la librería Controllers, y se ha hecho un estudio experimental de la eficiencia de estas implementaciones. Los resultados muestran que no hay un claro vencedor, con diferencias en función de la aplicación, la máquina y la carga computacional de los casos de prueba. Sin embargo sí se han detectado ciertos patrones.

Este trabajo también describe la implementación para GPU de la factorización de Cholesky, uno de los ejemplos más usados por el Centro de Supercomputación de Barcelona en sus demostraciones de OmpSs, en vistas a una futura implementación y comparativa con Controllers.

Abstract

Nowadays, the use of high-performance coprocessors in general purpose computing systems is commonplace, as reflected by the TOP500 list of the world's most powerful supercomputers. Programming these coprocessors efficiently presents a series of challenges, one of them is memory management. Coprocessors have their own memory space, and data transfers between the host machine and the coprocessor are costly.

In this context, numerous proposals for alleviating this problem arise, some of them native, such as CUDA's asynchronous data transfers or unified memory, and third party tools as well, which often aim to ease the use of these native resources for the programmer.

In this work, a comparison is made between two of these solutions: The programming model "Controllers", developed by the Trasgo group, and the OmpSs model, from the Barcelona Supercomputing Center. Though not their only aim, both present the programmer with an automatic system for introducing efficient data transfers.

For this comparison, an OmpSs implementation of the Controllers benchmarks used by the Trasgo group was made. The efficiency of these applications was later determined experimentally. The results show that OmpSs is, in most cases, less efficient than Controllers, although in different magnitude depending on the application and machine used.

This work also describes the GPU implementation of the Cholesky factorization, one of the prime examples used by the Barcelona Supercomputing Center in their OmpSs demonstrations, in view of a future Controllers version based on it.

Tabla de Contenidos

1 Introducción **1**

1.1 Contexto . . . 1

1.2 Motivación . . . 1

1.3 Objetivos . . . 2

1.4 Estructura del documento . . . 2

2 Planificación **5**

2.1 Planificación del proyecto . . . 5

2.2 Plan de contingencia . . . 6

2.3 Presupuesto . . . 6

3 Estado del arte y herramientas empleadas **9**

3.1 El modelo Controllers . . . 9

3.2 El modelo OmpSs . . . 9

3.2.1 Historia . . . 9

3.2.2 Modelo OmpSs . . . 10

3.2.3 Modelo de ejecución . . . 11

3.2.4 Ejemplos . . . 11

3.2.5 Algunas cláusulas importantes . . . 17

3.2.6 Diferencias con OmpSs-1 . . . 17

3.3 Herramientas Empleadas . . . 18

3.3.1 Grafo de tareas . . . 18

3.3.2 Extrae y Paraver . . . 18

3.3.3 Nvprof y Nvvp . . . 18

4 Descripción del problema 21

4.1 Matrixpow 21

4.1.1 Pseudocódigo 22

4.2 Hotspot 22

4.2.1 Pseudocódigo 23

4.3 Sobel 23

4.3.1 Pseudocódigo 24

4.4 Cholesky 24

4.5 Propuesta de solución 24

4.5.1 Matrixpow 24

4.5.2 Hotspot 25

4.5.3 Sobel 25

5 Implementación 27

5.1 Matrixpow 27

5.1.1 Versión inicial 27

5.1.2 Versión naïve 28

5.1.3 Primera versión con dependencias 28

5.1.4 Segunda versión con dependencias 28

5.1.5 Tercera versión con dependencias 30

5.1.6 Error de memoria 31

5.2 Hotspot 31

5.2.1 Primeros casos de prueba 31

5.2.2 Versión inicial 32

5.2.3 Primera versión con dependencias 32

5.2.4 Segunda versión con dependencias 32

5.2.5 Versión con solapamiento en GPU 33

5.2.6 Tercera versión con dependencias 33

5.3 Sobel 34

5.3.1 Versión inicial 34

5.3.2 Primera versión con dependencias 34

5.3.3 Segunda versión con dependencias 35

5.3.4 Tercera versión con dependencias 36
5.3.5 Cuarta versión con dependencias 36
5.3.6 Versión con solapamiento en GPU 36
5.4 Cholesky 37
5.4.1 Generación de resultados de referencia 37
5.4.2 Modificaciones a las estructuras internas del programa 37
5.4.3 Primera versión con GPU 38
5.4.4 Versión secuencial con GPU 38
5.4.5 Primeras versiones de Ompss 38
5.4.6 Versión con streams 39
5.5 Problemas de implementación y de despliegue 39
5.6 Conclusiones 41

6 Experimentación **43**
6.1 Objetivo de la experimentación 43
6.2 Metodología de experimentación 43
6.2.1 Escenario de experimentación 43
6.3 Resultados 44
6.3.1 Matrixpow 44
6.3.2 Hotspot 46
6.3.3 Sobel 48
6.4 Conclusiones 50

7 Conclusiones **51**
7.1 Objetivos cumplidos 52
7.2 Trabajo futuro 52
7.3 Valoración personal 53

Bibliografía **55**

Apéndices **59**

A Contenidos del Fichero ZIP **59**

Lista de Figuras

2.1 Planificación inicial del proyecto 5

3.1 Diagrama de la arquitectura de Controllers 10
3.2 Ejemplo del uso de dependencias entre tareas 12
3.3 Grafo de tareas para el programa de la figura 3.2 13
3.4 Ejemplo de una reducción usando tareas 14
3.5 Grafo de tareas para el programa de la figura 3.4 14
3.6 Código de host de la aplicación 15
3.7 Fichero de cabecera con la declaración del kernel 15
3.8 Código del kernel 16
3.9 Grafo de tareas para el programa de la figura 3.6 16

4.1 Pseudocódigo de Matrixpow 22
4.2 Pseudocódigo de Hotspot 23
4.3 Pseudocódigo de Sobel 24
4.4 Grafo de tareas de la factorización de Cholesky 26

5.1 Resultados en el profiler para la version de OmpSs 29
5.2 Resultados en el profiler para la nueva versión 30
5.3 Hotspot con entrada 2500 4 500 1 40
5.4 Matrixpow con entrada 1024 40 0 40
5.5 Sobel con entrada 100 40

6.1 Tiempos de ejecución de Matrixpow en Manticore 44
6.2 Tiempos de ejecución de Matrixpow en Pegaso 45
6.3 Tiempos de ejecución de Hotspot en Manticore 46

6.4 Tiempos de ejecución de Hotspot en Pegaso 47

Lista de Tablas

2.1 Plan de contingencia del proyecto 7
2.2 Presupuesto del proyecto 7

6.1 Porcentajes de cambio respecto a OmpSs en Matrixpow + Manticore 45
6.2 Porcentajes de cambio respecto a OmpSs en Matrixpow + Pegaso 46
6.3 Porcentajes de cambio respecto a OmpSs en Hotspot + Manticore 47
6.4 Porcentajes de cambio respecto a OmpSs en Hotspot + Pegaso 48
6.5 Tiempos de ejecución de Sobel en Manticore 48
6.6 Porcentajes de cambio respecto a OmpSs en Sobel + Manticore 49
6.7 Tiempos de ejecución de Sobel en Pegaso 49
6.8 Porcentajes de cambio respecto a OmpSs en Sobel + Pegaso 49

Capítulo 1

Introducción

1.1 Contexto

Como refleja la lista TOP500 [8] de los mayores supercomputadores del mundo, la configuración de muchos de ellos incluye coprocesadores de alto rendimiento, también conocidos como aceleradores hardware. Estos aceleradores pueden ser de varios tipos, como Unidades de Procesamiento Gráfico (GPUs) [23], o FPGAs. Programar este tipo de dispositivos de forma eficiente sigue siendo complicado.

Muchos modelos de programación para coprocesadores 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 el paralelismo reflejado en el código del kernel.

Por lo general, los kernels necesitan un conjunto de datos sobre los que trabajar. Estos datos han de ser transferidos entre la memoria del host (La máquina donde está alojado el coprocesador) y el coprocesador. Una forma de minimizar los sobrecostes asociados con múltiples transferencias de este tipo es realizar transferencias de un gran volumen de datos, que muevan en una sóla comunicación toda la información necesaria. En la ejecución de muchos programas encontramos transferencias de memoria y llamadas a kernels intercaladas. Realizar estas operaciones de forma secuencial introduce en ocasiones, importantes retrasos en la ejecución que podrían ser aliviados de forma significativa.

Muchos modelos de programación implementan el concepto de las transferencias de datos asíncronas para mitigar este problema. Este tipo de operaciones permiten que el host no lance las operaciones en GPU de forma secuencial, sino que éstas se ejecuten simultáneamente. El mayor inconveniente en el uso de estas llamadas es que obligan al programador a usar mecanismos complejos de bajo nivel, como streams y eventos, que en muchas ocasiones son, además, específicos del modelo de programación que se está usando, limitando la portabilidad de los programas a otro tipo de aceleradores. Por otro lado, detectar e implementar correctamente las situaciones donde se puede obtener un beneficio con estas técnicas requiere un análisis meticuloso de la aplicación y complica el código.

1.2 Motivación

Controllers [5] es un modelo de programación desarrollado por el grupo de investigación Trasgo [7] como una forma de simplificar la programación con aceleradores, proporcionando una forma sencilla de enviar kernels a distintos tipos de aceleradores, automatizando operaciones como la selección de los atributos de lanzamiento del kernel o las transferencias de datos entre host y acelerador.

Más recientemente, para simplificar el uso de transferencias de memoria asíncronas [9] en aplicaciones que puedan beneficiarse de ello, el grupo Trasgo propone una nueva solución basada en el modelo Controllers [6] para automatizar la introducción de estas operaciones. Esta solución evita que el programador tenga que conocer los aspectos más complejos del funcionamiento del coprocesador con el que está trabajando, y también ahorra un análisis en detalle del código, detectando de forma automática en tiempo de ejecución cuándo es posible solapar operaciones.

Al contrario que otros modelos que ofrecen este tipo de automatización, Controllers evita el uso de un planificador de tareas o técnicas de análisis de grafos que requieran un análisis automático de la aplicación, las cuales pueden llegar a introducir grandes sobrecostes.

Otro modelo de programación que introduce transferencias de datos automáticas de forma transparente para el programador es OmpSs [10], desarrollado por el Centro de Supercomputación de Barcelona (BSC) [13]. OmpSs es un modelo basado en tareas, donde las dependencias entre las mismas determinan su orden de ejecución. OmpSs, en su primera versión, permite al programador indicar qué datos hay que transferir al coprocesador antes de ejecutar una tarea determinada, y utiliza su sistema de dependencias para determinar cuándo se puede realizar la transferencia.

Este proyecto realiza una comparativa entre ambas soluciones, para determinar cuál de las dos ofrece una mayor eficiencia. Para ello, se desarrollan y prueban una serie de benchmarks con los que se realiza la comparativa.

1.3 Objetivos

El grupo Trasgo usa principalmente tres aplicaciones para determinar la eficiencia de Controllers: Matrixpow, Hotspot y Sobel. Estas aplicaciones cubren varios escenarios que se pueden dar en la computación paralela, por lo que son una buena representación de casos de uso reales para estos modelos.

Estos escenarios incluyen una aplicación donde el coste de los kernels de GPU es mucho mayor que el de las transferencias de datos, otra en la que sucede lo contrario, y las transferencias de datos obtienen una mayor relevancia, y por último una aplicación donde ambos costes son similares, además de introducir comunicaciones bidireccionales.

Para comprobar las diferencias entre OmpSs y Controllers se propone obtener versiones de las aplicaciones implementadas con OmpSs, y comparar los resultados obtenidos de esta forma con los de Controllers.

También se propone el desarrollo de una versión con GPU de la factorización de Cholesky [11], uno de los benchmarks más usados por el BSC en demostraciones de OmpSs, de cara a obtener una implementación con Controllers en el futuro. Esta aplicación no involucra transferencias de datos constantes, pero tiene características de interés en computación paralela como una carga computacional variable a lo largo de la ejecución de la aplicación, o un patrón de acceso irregular a los datos que usa.

Los objetivos del proyecto son los siguientes:

- Estudiar el modelo OmpSs.
- Desarrollar versiones de los benchmarks de Matrixpow, Hotspot y Sobel correctamente paralelizadas con OmpSs.
- Implementar la aplicación Factorización de Cholesky con el modelo de programación CUDA para ser ejecutada en GPUs de Nvidia.
- Experimentar con los benchmarks mencionados en ambos entornos, OmpSs y Controllers.
- Analizar los resultados obtenidos y extraer conclusiones.
  - Encontrar posibles puntos fuertes y limitaciones del modelo OmpSs.

1.4 Estructura del documento

El resto del documento se divide en los siguientes capítulos: El capítulo 2 describe la planificación inicial del proyecto y el coste de realización del mismo, el capítulo 3 introduce varios conceptos de estado del arte, en especial el funcionamiento del modelo OmpSs. El capítulo 4 describe los programas a implementar en Ompss y la paralelización que se pretende obtener de cada uno de ellos. El capítulo 6 describe la implementación de cada una de las aplicaciones y alguno de los principales problemas de despliegue de

las mismas. Por último, el capítulo 8 muestra los resultados de la experimentación realizada, y el capítulo 9 resume las conclusiones obtenidas y objetivos cumplidos.

Capítulo 2

Planificación

En este capítulo se presenta la planificación inicial del proyecto, posteriormente se resumen los cambios más importantes que sufrió la planificación, y por último se presenta el coste estimado del desarrollo del proyecto.

2.1 Planificación del proyecto

Inicialmente, se plantea un desarrollo iterativo de las cuatro aplicaciones mencionadas, donde se tendrá un período de tiempo para trabajar exclusivamente en cada una de las aplicaciones. También se planifican dos períodos para la instalación y aprendizaje de OmpSs, y uno para la realización de pruebas y comparativas con Controllers una vez finalizada la implementación. A continuación se muestra el diagrama de Gantt con la planificación inicial.

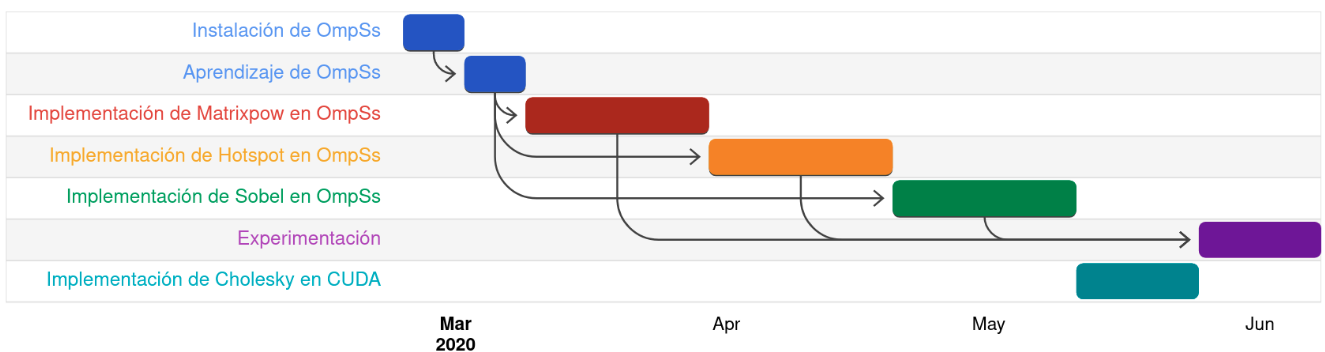

Figura 2.1: Planificación inicial del proyecto

Se planifica una semana para la instalación y pruebas en el cluster de OmpSs, y una semana más para el aprendizaje del modelo. Se asignan tres semanas para la implementación de cada una de las aplicaciones, a excepción de Cholesky, al considerarse una modificación algo más sencilla. Por último, se planea emplear dos semanas para realizar la experimentación con los benchmarks.

Se establece el orden de implementación de los benchmarks que aparece en la figura 2.1, pero al no existir dependencias entre estas tareas, podrían realizarse en cualquier orden. Lo mismo ocurre con la implementación de la factorización de Cholesky, que podría realizarse en cualquier punto del desarrollo del proyecto, al no tener dependencias con otras tareas. Es por esto que, de acuerdo con el método CPM podríamos definir tres caminos críticos:

- Instalación-Aprendizaje-Implementación de Matrixpow-Experimentación
- Instalación-Aprendizaje-Implementación de Hotspot-Experimentación

- Instalación-Aprendizaje-Implementación de Sobel-Experimentación

Al haber un único desarrollador, se han de realizar todas las tareas de forma lineal.

Debido a la naturaleza experimental del proyecto, no se siguió, en muchos casos, esta planificación inicial. Se realizaron múltiples versiones de cada una de las aplicaciones, donde con frecuencia fué necesario hacer cambios o volver atrás. En varias ocasiones se avanzó simultáneamente en las tres aplicaciones, al descubrir soluciones o mejoras aplicables a todas ellas. El tiempo empleado en esto también aumentó al descubrir errores y comportamientos inesperados de OmpSs, que se estudiaron más en detalle antes de continuar con la implementación de las aplicaciones.

La implementación en CUDA de Cholesky, por otro lado, se ajustó más a lo planeado, y se emplearon dos semanas en realizar las diferentes versiones y obtener el resultado deseado, aunque se comenzó dos semanas más tarde. La experimentación se realizó en varias partes, ya que fué necesario repetir algunas de las pruebas tras detectar problemas con las máquinas del cluster.

2.2 Plan de contingencia

La tabla 2.1 describe un plan de contingencia para los sucesos que pueden ocurrir a lo largo del desarrollo del proyecto.

2.3 Presupuesto

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

- Sueldo del desarrollador: estimamos el sueldo de un ingeniero informático junior en unos 50.000€ 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.

- Sueldo de los tutores: estimamos el sueldo de los tutores en unos 100.000€ brutos al año. Realizando el cálculo anterior, tenenos un coste de 50€ por cada hora de consultoría.

- Coste de las máquinas usadas en el cluster: se han utilizado principalmente dos máquinas, Manticore y Pegaso, con costes alrededor de 40000€ y 5000€ respectivamente. El tiempo de vida de estas máquinas se estima en unos 8 años, lo que nos deja con costes de 0.57€ y 0.07€ por hora, respectivamente.

- Coste de la máquina del desarrollador: la máquina utilizada tuvo un coste de 650€, con vistas a ser utilizada durante 6 años. El coste por hora es de 0.01€.

Las horas de utilización de cada recurso se estiman como:

- Desarrollador: 4 horas diarias durante 15 semanas, con 5 días laborables por semana. 300 horas.

- Tutores: 15 reuniones semanales de 1 hora, se estima que uno de los tutores estará presente en todas las sesiones mientras que el otro sólo en la mitad. Lo mismo ocurre para la resolución de dudas mediante correo electrónico, donde se estiman 10 y 5 horas respectivamente. 37.5 horas.

- Máquina Manticore: se usará durante la fase de implementación de las aplicaciones, así como durante la experimentación. Se estima que durante la fase de implementación, un 25% del tiempo dedicado al proyecto se empleará en pruebas de las aplicaciones, mientras que durante la experimentación será un 75% del tiempo. 85 horas.

- Máquina Pegaso: sólo se planea usar esta máquina durante la experimentación. Se deben realizar las mismas pruebas en ambas máquinas, por lo que la utilización sigue siendo de un 75% del tiempo. 30 horas.

- Máquina del desarrollador: se utilizará durante todo el proyecto. 300 horas.

En la tabla 2.2 se muestran las horas estimadadas de utilización de cada recurso, su coste en función de lo descrito anteriormente, y el coste total estimado para el proyecto.

Suceso	**Plan de Contingencia**
Avería de la máquina del cluster usada para el desarrollo (Manticore)	Se continúa el desarrollo en Pegaso, otra de las máquinas del cluster
Avería del cluster durante el desarrollo	Se continúa el proceso de implementación en la máquina 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
Imposibilidad de instalar alguna de las herramientas necesarias para el funcionamiento de OmpSs en el cluster	Uso de los recursos de otro centro de supercomputación accesible para el grupo/Uso de la máquina personal del desarrollador
Retraso en la implementación de las aplicaciones de OmpSs	Se retrasan la implementación de la factorización de Cholesky y la experimentación siempre que no afecte a la fecha de entrega del proyecto
Retraso en la implementación de las aplicaciones de OmpSs que afecta a la fecha de entrega del proyecto	Se elimina la implementación de la factorización de Cholesky del proyecto
No se tiene una paralelización satisfactoria de alguna de las aplicaciones en la fecha límite para el comienzo de la experimentación	Se realiza la experimentación con la paralelización parcial, teniéndolo en cuenta a la hora de analizar los resultados
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 git 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
Un error en OmpSs-2 impide implementar correctamente una o varias de las aplicaciones	Se implementan esas aplicaciones usando OmpSs-1

Tabla 2.1: Plan de contingencia del proyecto

Recurso	**Horas**	**Coste por hora**	**Coste**
Desarrollador	300	25€	7500€
Consultores	37.5	50€	1875€
Manticore	85	0.57€	48.45€
Pegaso	30	0.07€	2.1€
Máquina del desarrollador	300	0.01€	3€
		Total:	**9428€**

Tabla 2.2: Presupuesto del proyecto

Capítulo 3

Estado del arte y herramientas empleadas

Este capítulo describe los modelos Controllers y OmpSs, así como las herramientas empleadas durante el desarrollo del proyecto.

3.1 El modelo Controllers

Controllers [5], del grupo Trasgo es un modelo de programación paralela que surge para simplificar la tarea de los desarrolladores en un contexto como el actual, donde existen una gran variedad de aceleradores y modelos de programación para los mismos. Controllers permite definir kernels genéricos, que pueden lanzarse en distintos tipos de dispositivos, o kernels más especializados para un tipo concreto de dispositivos, y lanzarlos en el acelerador deseado de forma transparente para el programador.

Estos lanzamientos transparentes incluyen realizar de forma automática la selección de los parámetros de lanzamiento del kernel, a diferencia de otros modelos con propósitos similares, como OpenACC [14], y transferir automáticamente los datos necesarios entre host y acelerador.

El modelo se basa en la entidad abstracta "Controller". Estas entidades mantienen una cola de kernels a ejecutar, así como una lista de estructuras de datos asociadas a los mismos. Desde el host se envían los kernels a ejecutar a la cola de un Controller, y este decide en función de una política de ejecución cuándo enviarlo al dispositivo al que está asociado. Son los Controllers los que gestionan la selección de los parámetros de lanzamiento apropiados, y la gestión de datos, realizando transferencias en función de las estructuras de datos asociadas al kernel.

Más recientemente, el grupo Trasgo propone una ampliación al modelo Controllers [6] para proporcionar una forma sencilla de introducir solapamiento entre cómputo y transferencias de datos entre host y acelerador.

Al contrario que otros modelos donde se usan planificadores de tareas o técnicas de análisis de grafos, Controllers incluye la información sobre dependencias de datos en las estructuras de datos asociadas a los kernels, y realiza las sincronizaciones necesarias en base a esta información. Así, se analiza la secuencia de operaciones en tiempo de ejecución, y se solapan de forma automática transferencias de datos y cómputo.

Este modelo permite que el programador no tenga que lidiar con aspectos más complicados de la implementación con aceleradores como streams o eventos, y evita errores que se puedan cometer al incluir estos mecanismos en la aplicación.

3.2 El modelo OmpSs

3.2.1 Historia

OmpSs [10] es un modelo de programación paralela desarrollado a partir de los modelos OpenMP [12] y StarSs por el Centro de Supercomputación de Barcelona. OpenMP nos proporciona una forma de

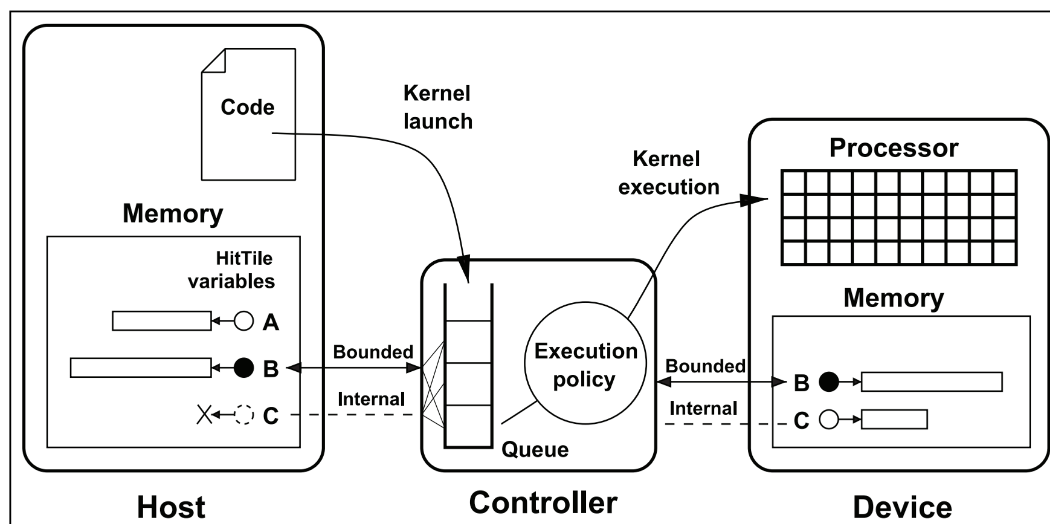

Figura 3.1: Diagrama de la arquitectura de Controllers obtenido del artículo [5] del grupo Trasgo. Las peticiones de lanzamiento de kernels se envían a la entidad Controller, que gestiona su ejecución y la transferencia de las variables "Bounded" entre espacios de memoria. Podemos ver la diferencia entre la variable A, definida sólo en el host, la B, asociada a una serie de kernels y que por tanto se transfiere automáticamente entre host y dispositivo, y la C, una variable interna del Controller que se define en el host pero para la que sólo se reserva memoria en el dispositivo.

obtener una aplicación paralela a partir de una versión secuencial, añadiendo anotaciones al código. Estas anotaciones indican cómo se compartirá el trabajo entre hilos, y permiten al compilador producir una versión paralela de la aplicación. StarSs también se basa en introducir anotaciones en el código, pero tiene importantes diferencias con OpenMP:

- Se usa una pool de hilos durante toda la ejecución del programa, en lugar de la solución de OpenMP, donde se crean hilos al inicio de cada región paralela.
- StarSs permite programar para sistemas heterogéneos, mientras que OpenMP se centra en sistemas de memoria compartida.
- Se introduce el concepto de tarea: Bloques de código que pueden ejecutarse de forma asíncrona en paralelo.
- Se introduce un mecanismo de dependencias para controlar de forma automática el orden de ejecución de las tareas, evitando la necesidad de sincronizar explícitamente partes de la ejecución.

El objetivo de OmpSs es introducir los cambios que OpenMP necesita para usarse en las arquitecturas más modernas. Existen dos generaciones del modelo, con importantes diferencias entre ambas en algunas áreas. La descripción que aparece en este documento se refiere a OmpSs-2, ya que es el modelo que se ha utilizado en el proyecto, aunque se indican las diferencias más importantes con la versión anterior.

3.2.2 Modelo OmpSs

OmpSs es un modelo de programación que pretende extender OpenMP con nuevas directivas para permitir paralelismo asíncrono e introducir soporte para aceleradores. Se compone de una serie de librerías y directivas que se pueden utilizar junto a programas escritos en C, C++ o Fortran para producir aplicaciones concurrentes.

El objetivo del modelo es permitir, a través de una serie de anotaciones en el código fuente del programa, que el compilador genere una versión paralela del código. Éstas anotaciones permiten definir bloques de código como tareas, cuya ejecución es gestionada por un planificador interno. Es posible

determinar dependencias entre tareas, utilizando una serie de directivas que permiten declarar qué datos necesita una tarea antes de comenzar su ejecución, y cuáles da como salida.

La implementación de OmpSs consta de un compilador de fuente a fuente y una librería de tiempo de ejecución. El compilador Mercurium [18], genera una versión paralela de la aplicación, que después es compilada en un ejecutable por el compilador que elija el programador. La librería, Nanos [19] [20], proporciona al programa una serie de servicios como creación de tareas, gestión de dependencias, etcétera.

3.2.3 Modelo de ejecución

Al inicio de la ejecución, se crea un conjunto de hilos inicial. La función main se considera una tarea implícita, y se añade al planificador de tareas como "lista". Uno de los hilos creados comienza la ejecución de esta función.

Cuando un hilo en ejecución llega a una sección del código que ha sido declarada como tarea, en lugar de ejecutarla, la introduce en una cola del planificador de tareas. Esta tarea será asignada a uno de los hilos disponibles para el programa, pudiendo comenzar antes o después en función de la disponibilidad de los hilos y de las restricciones asociadas a la tarea.

Dependencias

En la declaración de cada tarea se puede indicar, a través de una série de directivas, qué datos necesita la tarea antes de comenzar su ejecución, así como qué datos produce como salida.

Durante la ejecución, se utilizan las dependencias asociadas a cada tarea y el orden en el que se han ido creando para obtener una serie de restricciones del orden de ejecución de las mismas. Cada vez que se crea una tarea, se contrastan sus dependencias con las de las tareas que ya están en el planificador, y se introduce en un grafo. El planificador de tareas decide de esta forma cuándo una tarea está lista para su ejecución: Cuando todos sus predecesores en el grafo han terminado.

Además de dependencias de entrada y salida, se puede indicar que una tarea puede ejecutarse de forma concurrente a otras tareas que usen los datos indicados, y, al contrario, que una tarea debe ejecutarse sólo cuando no haya otras tareas usando los datos indicados.

Por lo general, las dependencias de una tarea se liberan cuando esta termina, con la excepción de las que tienen tareas hijas. Sólo se pueden establecer dependencias entre tareas con el mismo progenitor, de forma que si se quieren establecer dependencias con la cláusula hija de una tarea anterior, ésta tarea padre debe declarar también las dependencias de la misma. Cuando termine la tarea padre, se liberarán sus dependencias a excepción de las que también haya declarado su hija.

Para este tipo de situaciones también existen las dependencias débiles: Dependencias que sólo necesitan las tareas hijas. Declarar una dependencia de entrada débil permite que una tarea comience su ejecución sin necesidad de esperar por tareas anteriores que modifiquen esos datos. La espera se realiza sólo en las tareas hijas que necesiten los datos.

3.2.4 Ejemplos

A continuación se incluyen algunos ejemplos de código escrito en OmpSs, junto con el grafo de tareas que generan.

Uso de dependencias

El código en la figura 3.2 define 5 tareas de OmpSs, con dependencias de entrada y salida sobre tres variables. Las dependencias se declaran en función de lo que haga la tarea con las variables, así, por ejemplo, en la tarea 1 se declara una dependencia de entrada y otra de salida sobre la variable x con la

cláusula inout, ya que modificamos la variable dentro de la tarea, pero en la tarea 4 sólo declaramos una dependencia de entrada sobre x, ya que utilizamos su valor dentro de la tarea pero no se modifica.

Se establecen dependencias entre la tarea uno y la dos por la variable x, entre la dos y la tres por la variable x, entre la dos y la cuatro por la variable y, entre la tres y la cuatro por la variable x, y por último entre la cuatro y la cinco por la variable z. Esto garantiza que estas tareas, que de no tener dependencias podrían ejecutarse en el momento en el que entraran a la cola del planificador, se ejecuten en el orden deseado. En la figura 3.3 vemos una representación gráfica de estas dependencias.

```
#include <stdio.h>

int main(int argc, char *argv[]){
        int x;
        int y;
        int z;

        #pragma oss task inout(x) shared(x) label(task 1)
        {
                x++;
        }
        #pragma oss task inout(x, y) shared(x, y) label(task 2)
        {
                x++;
                y++;
        }
        #pragma oss task inout(x) shared(x, y) label(task 3)
        {
                x++;
        }
        #pragma oss task in(x, y) out(z) shared(x,y) label(task 4)
        {
                z = x+y;
        }
        #pragma oss task in(z) label(task 5)
        {
                printf("%d\n", z);
        }

        #pragma oss taskwait
        return 0;
}
```

Figura 3.2: Ejemplo del uso de dependencias entre tareas

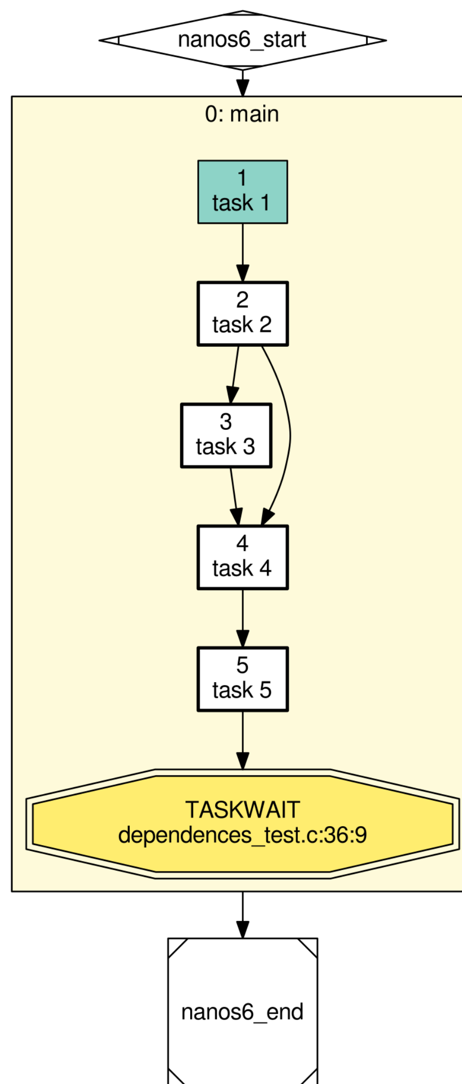

Figura 3.3: Grafo de tareas para el programa de la figura 3.2

Reducción usando tareas

En la aplicación de la figura 3.4, basada en uno de los ejemplos mostrados en la guía de usuario de OmpSs-2 [15], se realiza una reducción: Una operación usada en programación paralela para obtener un resultado a partir de los elementos de un array. En este caso, se realiza la suma de todos los elementos del array, dividiendo el trabajo entre varios hilos. En la aplicación, se declara el contenido del bucle for como una tarea, dentro de la misma, se añade a la variable sum, compartida por todas las tareas lanzadas de esta forma, el valor de la posición correspondiente del array. El incremento de la variable sum se realiza en exclusión mútua por todas las tareas gracias a la cláusula atomic. De nuevo, se puede ver una representación gráfica de esta aplicación en la figura 3.5, en este caso no se establecen dependencias entre las tareas, y todas pueden ejecutarse simultáneamente.

```
#include <stdio.h>
#include <stdlib.h>
#define N 5

int main(int arcg, char *argv[]){
    for(int i=0; i<N; i++){
        a[i]=i;
    }
    int sum = 0;
    for(int i=0; i<N; i++){
        #pragma oss task shared(sum) label(Reduccion)
        {
            #pragma oss atomic
            sum+=a[i];
        }
    }
    #pragma oss taskwait
    printf("sum:␣%d\n", sum);

    return 0;
}
```

Figura 3.4: Ejemplo de una reducción usando tareas

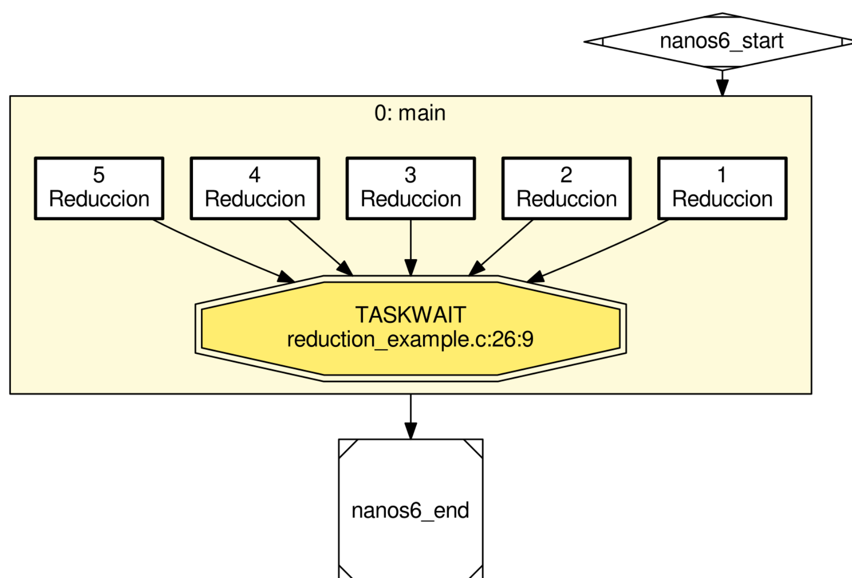

Figura 3.5: Grafo de tareas para el programa de la figura 3.4

Uso de tareas de CUDA

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include "cuda.h"
#define N 5

int main(int argc, char *argv[]){
    int n=N;
    int* a = (int*)malloc(N * sizeof(int));
    for(int i=0; i<N; i++){
        a[i]=0;
    }
    int* d_a;
    cudaMalloc((void**)&d_a, N * sizeof(int));

    #pragma oss task out(d_a[0;n]) label(Copia HtD)
    cudaMemcpy(d_a, a, N*sizeof(int), cudaMemcpyHostToDevice);

    kernel_example(d_a, N);

    #pragma oss task in(d_a[0;n]) out(a[0;n]) label(Copia DtH)
    cudaMemcpy(a, d_a, N*sizeof(int), cudaMemcpyDeviceToHost);

    int sum = 0;
    for(int i=0; i<N; i++){
        #pragma oss task in(a[i]) shared(sum) label(Reduccion)
        {
            #pragma oss atomic
            sum+=a[i];
        }
    }
    #pragma oss taskwait
    printf("sum:␣%d\n", sum);

    return 0;
}
```

Figura 3.6: Código de host de la aplicación

```
#pragma oss task inout(a[0;N]) ndrange(1, N, N) device(cuda) label(kernel)
__global__ void kernel_example(int* a, int N);
```

Figura 3.7: Fichero de cabecera con la declaración del kernel

```
#include "cuda.h"
#include <stdio.h>

__global__ void kernel_example(int* a, int N){
        a[threadIdx.x]++;
}
```

Figura 3.8: Código del kernel

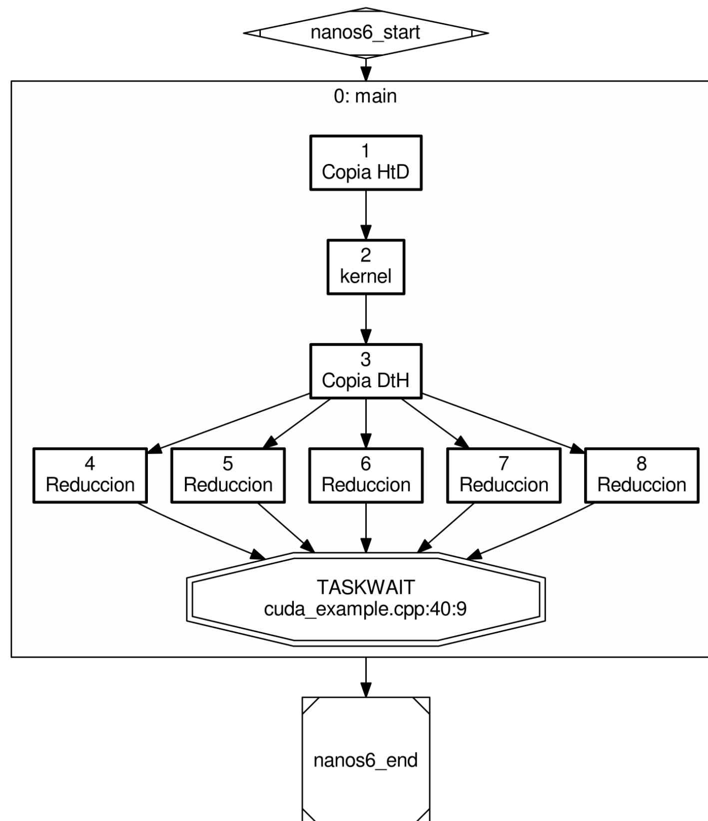

Figura 3.9: Grafo de tareas para el programa de la figura 3.6

Por último, el ejemplo en la figura 3.6 muestra una aplicación similar a la mostrada en la figura 3.4, pero donde la inicialización del array se realiza a través de un kernel de GPU. Para declarar kernels de CUDA como tareas de OmpSs primero debemos separar un archivo de cabecera y un archivo con el propio código del kernel del programa principal. Estos archivos han de ser compilados por separado con el compilador de Nvidia y enlazados después con el main.

En la declaración del kernel vemos, además de una directiva que indica que la tarea se ejecutará en GPU, la directiva ndrange. Esta directiva es usada para determinar el tamaño de los bloques y de grid en GPU en OmpSs, a diferencia de la forma habitual, donde indicamos estos tamaños en la llamada al kernel. Ndrange toma un primer argumento con el número de dimensiones de un bloque, n, después, toma n argumentos indicando el tamaño de cada una de las dimensiones, por último, toma n argumentos indicando el número de hilos que se desea lanzar en cada dimensión, esto se utiliza para determinar el tamaño de grid.

En la función main tenemos, por tanto, una tarea que engloba la copia del array a GPU, la llamada

al kernel, otra tarea que engloba la transferencia de los datos de vuelta al host, y por último el bucle que veíamos en la aplicación anterior. Las dependencias que se establecen entre ellas se muestran en la figura 3.9.

El modo de declarar las dependencias que se muestra en este ejemplo: array[inicio;longitud], nos permite declarar dependencias sobre varias posiciones de un array de forma sencilla, así, la directiva inout(a[0;N]) que vemos en el kernel declara dependencias sobre a[0], a[1], ..., a[N].

3.2.5 Algunas cláusulas importantes

En esta sección se incluyen algunas de las cláusulas de OmpSs que aparecerán con frecuencia más adelante:

- **Taskwait.** Las cláusulas taskwait en OmpSs son puntos de sincronización donde se interrumpe la ejecución de la tarea donde se encuentren, y se espera a que todas las tareas hijas de la misma hayan terminado antes de continuar la ejecución.

- **Commutative.** La cláusula commutative de OmpSs toma una variable como argumento. Las tareas que tienen en su declaración una cláusula commutative se ejecutan en exclusión mútua con otras tareas que tengan una cláusula commutative con el mismo valor, así como tareas que tengan dependencias sobre ese valor.

- **Task for.** La directiva task for toma un bucle y genera una serie de tareas, cada una de las cuales realiza parte de las iteraciones del bucle. El número de tareas generado puede ser automático, en función de los hilos disponible, o fijo, si se indica un tamaño de chunk a través de la directiva chunksize.

3.2.6 Diferencias con OmpSs-1

Una de las diferencias más significativas respecto a OmpSs-1, al menos en lo que concierne a este proyecto, es el cambio en el modelo de memoria. Por lo demás, OmpSs-2 incluye muchos de los cambios esperables en una nueva generación del modelo: Nuevas directivas, cambios en la sintáxis, mejoras en la eficiencia...

Algunos de estos cambios serían el hecho de que en OmpSs-2 todos los hilos son iguales, mientras que en OmpSs-1 existe un hilo maestro (el único que ejecuta el main), la inclusión de dependencias débiles en OmpSs-2 (sección 3.2.3), o que OmpSs-1 no libera las dependencias de una tarea hasta que todas sus hijas han terminado.

Cambio en el modelo de memoria

Una de las características más importantes de OmpSs-1 en cuanto a la programación de sistemas con aceleradores es que presenta al programador un único espacio de direcciones de memoria. Permite mover datos entre el host y el acelerador de forma automática, sin que el programador tenga que escribir el código para ello.

OmpSs-1 proporciona una serie de directivas, separadas de las dependencias, que permiten especificar, para una tarea que se tiene que ejecutar en un acelerador, qué datos hay que mover al mismo antes de comenzar la ejecución, y cuáles hay que llevar de vuelta al host cuando acabe.

Sin embargo, esta funcionalidad no existe en OmpSs-2. Esto no sólo lleva a que el programador tenga que escribir código específico para el acelerador que se está usando, sino que también es necesario lidiar con las sincronizaciones necesarias para asegurar que los datos estén disponibles en el espacio de memoria indicado antes de comenzar la ejecución de la tarea.

Otro cambio importante y relacionado con esto es que OmpSs-2 sólo da soporte a tareas de CUDA, al contrario que OmpSs-1, que permite ejecutar también código de OpenCL.

3.3 Herramientas Empleadas

3.3.1 Grafo de tareas

La librería Nanos [19] tiene una opción de ejecución que permite generar una representación gráfica en pdf del grafo de tareas usado por el planificador durante la ejecución del programa. Mediante esta representación es posible seguir paso a paso el descubrimiento de nuevas tareas, las dependencias que se establecen entre ellas, y cuáles están en ejecución en un momento dado. Existe una directiva que permite asignar nombres a las tareas, de forma que el análisis del grafo sea más sencillo.

Esta ha sido una de las herramientas de más utilidad durante el desarrollo del proyecto, por la simplicidad del análisis del grafo y la facilidad con la que se detectan ciertos problemas simples. Una desventaja del grafo, por otro lado, es que el análisis se hace más complicado en programas que generan muchas tareas, siendo la representación menos clara.

3.3.2 Extrae y Paraver

Extrae

Extrae [21] es un paquete de instrumentación dinámica desarrollado por el BSC. Permite generar trazas de las aplicaciones que posteriormente pueden ser visualizadas usando Paraver [22].

Extrae puede ser activado directamente con una serie de opciones por defecto a través de Nanos, en el caso de aplicaciones escritas en OmpSs. También incluye una serie de librerías que permiten generar trazas de programas que usen MPI, OpenMP, pthread, SMPs, CUDA, OpenCL y Java.

Es posible definir los datos que se deben incluir en la traza a través de un fichero XML. En el uso que se hizo del paquete para este proyecto, la configuración por defecto fué suficiente para obtener la información deseada. Algunas de las aplicaciones de esta información fueron visualizar las dependencias establecidas en ejecución, y comprobar el orden y tiempos de ejecución de las tareas.

Paraver

Paraver [22] es la herramienta del BCS para analizar las trazas generadas por Extrae. Proporciona una interfaz gráfica para visualizar los datos contenidos en la traza. Aunque permite combiar de forma manual diferentes factores para crear visualizaciones concretas, para un análisis general de la traza es suficiente con usar las configuraciones por defecto que proporciona el BSC.

Las funcionalidades más útiles durante este proyecto fueron la capacidad de visualizar en una línea de tiempo la ejecución de cada tarea, así como el procesador asignado al trabajo, la capacidad de visualizar las dependencias entre tareas en esta línea de tiempo, y el poder ver la duración de cada tarea, así como los tiempos de inicio y finalización.

De esta forma, durante el desarrollo de las aplicaciones del proyecto, se comprueba el orden de ejecución de cada directiva, y se detectan posibles ineficiencias en base a la información sobre hilos, streams, y la duración de cada llamada.

En comparación a la otra herramienta de visualización utilizada, nvvp, de Nvidia, se obtiene por lo general menos información y el uso es más complicado.

3.3.3 Nvprof y Nvvp

Nvprof [16] y Nvvp [17] son los equivalentes en CUDA de Extrae y Paraver. Nvprof permite generar la traza de una aplicación recogiendo datos relativos a las operaciones que involucran a la GPU, como los kernels o las transferencias de memoria. Asimismo, recoge información sobre los hilos de CPU asociados a estas operaciones.

Nvvp (Nvidia Visual Profiler), al igual que Paraver, nos permite visualizar los datos contenidos en la traza generada por Nvprof en una línea de tiempo. Aunque esta traza no muestra elementos propios de OmpSs, como las dependencias, resulta muy útil disponer de información sobre las operaciones en GPU.

Debido a la facilidad de uso de Nvvp y a que, por lo general, es más interesante visualizar los datos de las operaciones en GPU, estas son las herramientas más utilizadas durante el proyecto, aunque en muchas ocasiones en conjunto con Extrae y Paraver.

Capítulo 4

Descripción del problema

En este capítulo se da una descripción de las aplicaciones que se han implementado en OmpSs: Matrixpow, Hotspot y Sobel, así como la factorización de Cholesky, de la que se pretende obtener una implementación con CUDA.

4.1 Matrixpow

Esta aplicación realiza una cadena de multiplicaciones de matrices en GPU para obtener la enésima potencia de una matriz dada. La multiplicación de matrices es una operación que, dadas dos matrices A de dimensiones $l \times m$ y B de dimensiones $m \times n$ calcula C como $c_{ij} = \sum_{k=1}^{m} a_{ik} * b_{kj}$

Se obtuvo a partir de los benchmarks de multiplicación de matrices 2mm y 3mm de PolyBench [2]. Dado un entero, n, y una potencia, p, inicializa una matriz A de dimensiones $n \times n$ y realiza una cadena de multiplicaciones para obtener A^p. La salida del programa es la normalización de las matrices intermedias obtenidas en este proceso: $C_i = A^i : i \epsilon [1 : p]$.

Consta de dos partes, una sección con cómputo en GPU y otra con cómputo en host. La sección de GPU consiste en el cálculo de la multiplicación de la matriz original por el resultado parcial anterior: $C_k = C_{k-1} * C_0 : k \epsilon [1 : p]$ con $C_0 = A$. El kernel para la multiplicación de matrices proviene de los ejemplos del Toolkit de CUDA.

En la sección de host se calcula la normalización de los resultados parciales, obtenidos en cada iteración, y se almacena en un buffer. La norma p de un vector V es una función matemática que transforma el vector en un número real. La función de normalización elegida en este caso consiste en: (a) Determinar los valores máximo y mínimo en la matriz; (b) Restar a cada elemento de la matriz el mínimo, y dividir el resultado entre el máximo; (c) Calcular la raíz cuadrada de la suma de cada elemento de la matriz al cuadrado; (d) dividir cada elemento de la matriz entre este resultado.

En la figura 4.1 se muestra en pseudocódigo el bucle principal de la aplicación. Se usan tres matrices, A, B y C, que se reservan tanto en host como en GPU. El código distingue entre iteraciones pares e impares, y en cada una se utiliza una matriz de destino distinta. En cada iteración, se calcula un resultado parcial en GPU, se copia a la memoria del host, y por último se realiza la normalización en el host.

En esta aplicación, la duración de los kernels es considerablemente mayor que la de las transferencias de datos, como se puede ver en las figuras 5.1 y 5.2. Esto refleja un caso de uso distinto al de los otros benchmarks. En este caso, el solapamiento de transferencias de datos y cómputo podría ser menos relevante.

4.1.1 Pseudocódigo

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

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)
```

Figura 4.1: Pseudocódigo de Matrixpow

4.2 Hotspot

Esta aplicación simula la transmisión del calor en una superficie, se suele usar para estimar las temperaturas en procesadores. Obtenido de los benchmarks de Rodinia [1], esta aplicación se modificó para obtener un código que pueda explotar las ventajas de Controllers.

El programa original recibe como argumentos un número N, un número H "altura de la pirámide" y un número de iteraciones para la simulación. Después de inicializar una matriz de tamaño $N \times N$, se lanzan grupos de H iteraciones de actualización al acelerador hasta que se llega al número indicado.

En la modificación realizada por el grupo Trasgo, la aplicación recibe un argumento adicional, el número de iteraciones por copia "ipc". En este caso, cada ipc actualizaciones de la matriz se realiza un "copyback": se copia el estado de la superficie de vuelta al host. De esta forma, se introducen transferencias de datos que se pueden solapar con fases de cómputo. Un posible uso para estos datos en el host sería generar una representación gráfica del proceso de transmisión del calor.

En la figura 4.2 se puede ver el pseudocódigo para esta aplicación. De nuevo, al igual que en Matrixpow, tenemos un bucle donde las matrices de destino son distintas en iteraciones pares e impares. En una iteración se llama al kernel y, si se ha realizado un número suficiente de iteraciones, se realiza la transferencia del resultado al host. En el host se realiza la copia de este resultado a un buffer, simulando lo que podría hacer una aplicación real.

En Hotspot nos encontramos con un caso de uso diferente al de Matrixpow, donde las transferencias de datos llevan más tiempo que los kernels en GPU. Además de eso, es posible ajustar el número de transferencias que se realizan, lo que varía la carga computacional asociada a las mismas y por tanto afecta a la eficiencia de la paralelización realizada.

4.2.1 Pseudocódigo

```
n = número de iteraciones de la simulación
ipc = número de iteraciones por copia
h = altura de la pirámide
Declaración de matrices O y D en host y GPU
Declaración del buffer C

for i in 0..n:
    Intercambio de las matrices O y D
    llamada al kernel, realiza h iteraciones de actualización
    if((i % ipc) == 0):
        transferencia del resultado en GPU a la matriz D en host
        copia de D a C
    i = i+h
```

Figura 4.2: Pseudocódigo de Hotspot

4.3 Sobel

Esta aplicación recibe un vídeo en formato YUV como entrada y procesa de forma iterativa el número de fotogramas que se indique, aplicándoles el filtro de Sobel [24], un filtro convolucional normalmente usado para detectar bordes en una imágen. La implementación inicial se obtuvo en http://cuda-programming.blogspot.com/ [3].

De cara a una aplicación práctica, por ejemplo en sistemas de visión computacional, una aplicación de este tipo debe ser capaz de procesar vídeo en tiempo real, para lo cual es importante operar de la forma más eficiente posible.

El vídeo que procesa la aplicación contiene tres componentes por fotograma, cada uno de los cuales se procesa independientemente. En una iteración, como se puede observar en la figura 4.3, se transfiere cada uno de los componentes a la GPU, se procesa, y se transfiere de vuelta al host. Después, en el host, se carga en memoria cada uno de los componentes y se almacenan los resultados. Estos bucles de lectura y escritura están dentro de sendas funciones, en la implementación.

Existen cuatro versiones diferentes de Sobel, en función del orígen y el destino de los fotogramas: Fichero a Fichero, Fichero a Memoria, Memoria a Fichero y Memoria a Memoria. Esto permite experimentar con diferentes tiempos de lectura y escritura en el host.

En esta aplicación aparecen tiempos de ejecución muy similares para los kernels y las transferencias de memoria. Estos tiempos son también especialmente cortos, lo cual complica la implementación de una versión que solape estas comunicaciones de forma eficiente. Otra característica de este programa es que realiza comunicaciones bidireccionales durante la ejecución, mientras que Matrixpow y Hotspot sólo mueven datos de GPU a host.

4.3.1 Pseudocódigo

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

Figura 4.3: Pseudocódigo de Sobel

4.4 Cholesky

Calcula la factorización de Cholesky [11] de una matriz de tamaño $n \times n$ generada por la aplicación con las propiedades necesarias. La factorización de Cholesky es más eficiente que la factorización LU típicamente usada en sistemas informáticos para resolver sistemas de ecuaciones, con el inconveniente de que sólo es aplicable a matrices simétricas positivas-definidas. La implementación inicial de la aplicación proviene de los ejemplos de OmpSs-2 [25], y es uno de los benchmarks usados por el BSC en demostraciones de OmpSs.

Al contrario que en las anteriores aplicaciones, donde se realizan movimientos de datos constantes durante la ejecución, una implementación en GPU de este programa seguirá el esquema input-compute-output, donde los datos se copian a GPU al inicio del programa y no se realizan más movimientos hasta que la fase de cómputo termina, y los resultados se envían de vuelta al host.

Aunque no existe la posibilidad de solapar comunicaciones con cómputo, se consideró interesante implementar una versión en GPU de la aplicación como base para una futura implementación con Controllers debido a las características de este algoritmo. En la figura 4.4 se puede ver el grafo de tareas asociado a este programa, donde cada recuadro corresponde a una de las operaciones de álgebra lineal que se realizan. Podemos ver que la carga computacional varía durante la ejecución del programa, concentrándose el trabajo cada vez en menos tareas. Otra de las propiedades interesantes de la aplicación es un patrón de acceso irregular a la matriz que se está procesando, donde se realiza un mayor número de lecturas en ciertas zonas de la matriz.

4.5 Propuesta de solución

A continuación se resume la forma en que se pretende paralelizar cada uno de los benchmarks de Trasgo. Al tratarse de OmpSs-2, el programador tiene que decidir, además de qué secciones se pueden ejecutar en paralelo, cómo se deben solapar las transferencias de datos con otras operaciones.

Durante el proyecto se ha realizado un desarrollo iterativo de las versiones de OmpSs de las aplicaciones, con el objetivo de conseguir la paralelización aqui descrita.

4.5.1 Matrixpow

En el caso de matrixpow, es posible comenzar la ejecución del kernel de la iteración i cuando han terminado tanto el kernel de la iteración i-1 como la transferencia de los datos de la matriz de destino para la iteración i al host.

Por tanto, es posible solapar la transferencia de resultados y la normalización de la matriz de la iteración i-1 con el kernel de la iteración i, ya que no influyen en el mismo. Esto es lo que se pretende conseguir con la paralelización de esta aplicación.

4.5.2 Hotspot

Hotspot tiene una estructura similar a la de matrixpow, con la excepción de que no se realizan copias de memoria ni trabajo en el host en todas las iteraciones. Lo que se pretende alcanzar con la paralelización de esta aplicación es que la transferencia de resultados y función de host de la iteración i-1 se solape con la ejecución del kernel de la iteración i, en las iteraciones donde se realicen.

4.5.3 Sobel

En Sobel tenemos que, para cada fotograma, se ejecuta el mismo código tres veces: una por cada componente del mismo. Cada una de estas ejecuciones se puede realizar de forma independiente y en cualquier orden, no hay dependencias entre ellas. Por tanto, el primer objetivo en la paralelización de esta aplicación es conseguir que los tres componentes se procesen en paralelo.

Dentro del procesamiento de cada componente es posible solapar la carga del componente de la iteración i+1 con el kernel y transferencia de GPU a host de la iteración i. También se puede solapar la escritura del resultado de la iteración i-1 con el kernel y transferencia de host a GPU de la iteración i.

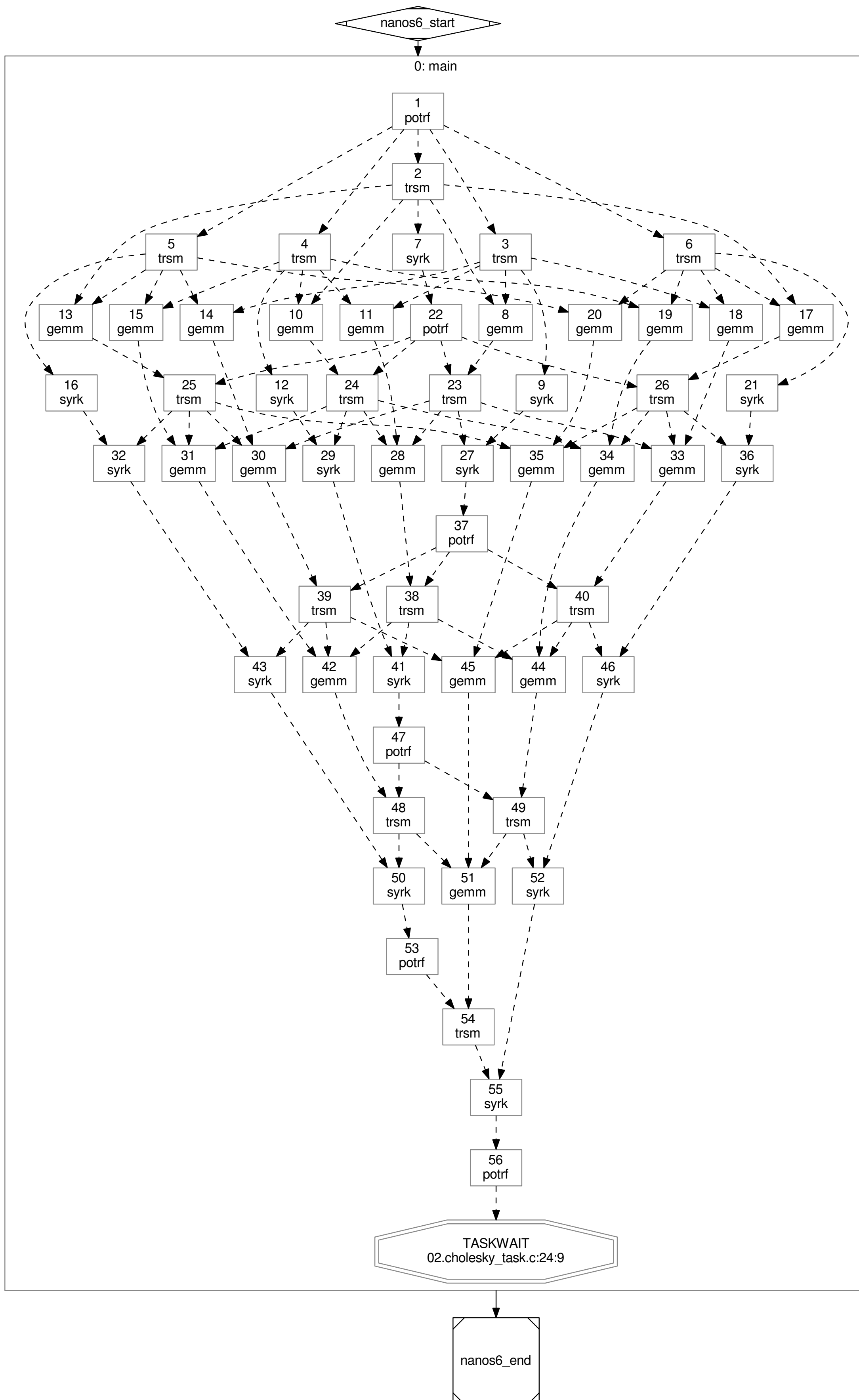

Figura 4.4: Grafo de tareas de la factorización de Cholesky

Capítulo 5

Implementación

En este capítulo se describe el proceso de implementación de las versiones con OmpSs de cada una de las aplicaciones. La implementación se ha realizado de forma iterativa, por lo que, para cada aplicación, se describe una serie de versiones por las que se ha pasado para llegar al resultado final. Al final del capítulo, se describe uno de los problemas que se encontraron al ejecutar aplicaciones compiladas con Mercurium en el cluster.

Durante el proceso de implementación se utiliza principalmente la máquina Manticore (Cuya configuración se describe en la sección 6.2.1) para probar las sucesivas versiones, por lo que todos los porcentajes de mejora que se indican en el capítulo corresponden a los tiempos de ejecución en esta máquina.

Para cada uno de los benchmarks del grupo Trasgo se dispone de dos versiones de referencia escritas en CUDA, una con transferencias de datos síncronas y otra con transferencias asíncronas. En todos los casos, la versión de referencia a partir de la que se desarrolla la de OmpSs es la síncrona.

5.1 Matrixpow

5.1.1 Versión inicial

Se modifica el código original de referencia para que el kernel se lance como una tarea de OmpSs. Para ello se divide el código en tres ficheros, uno con el código de host (Todo el código original a excepción del kernel de CUDA), otro con la declaración del kernel, donde también se declara el kernel como tarea de OmpSs, y un tercero con el código del kernel.

En OmpSs, el tamaño de los bloques de hilos en GPU, así como el tamaño del grid de bloques se indican a través de la cláusula **ndrange**, en el pragma que declara el kernel como una tarea. Para poder acceder a esos valores desde el pragma, se pasan al kernel los vectores dim3 que contienen esos tamaños en el Host.

Con esta versión inicial se obtienen tiempos de ejecución más lentos que los de la referencia en ejecuciones cortas, y algo más rápidos en las más largas. En el caso de las ejecuciones más rápidas (alrededor de 0.01 y 0.02 segundos respectivamente en la referencia), el tiempo de ejecución aumenta en alrededor de un 400%. En los tres casos de prueba siguientes (Alrededor de 0.25, 0.5, y 1.05 segundos en la referencia), el aumento está cerca de un 30%. Una hipótesis al respecto es que este aumento de tiempo se deba al planificador de tareas de OmpSs, al introducir un sobrecoste cuando, por lo demás, la aplicación es igual a la original.

Por último, en los tres casos de prueba finales (tiempos de referencia alrededor de 6, 12, y 28 segundos), se observan mejoras en torno al 10% para la versión con OmpSs.
Los resultados numéricos obtenidos en varios de los casos de prueba utilizados con esta aplicación difieren ligeramente de los obtenidos con la versión nativa. Los cambios son pequeños, cuando aparecen sólo cambian los últimos 2 o 3 decimales. La causa más probable para estas diferencias es un cambio en el orden en el que se realizan algunas operaciones de punto flotante. Se decide utilizar la salida de esta primera versión con OmpSs para comprobar la exactitud de las siguientes versiones.

5.1.2 Versión naïve

El bucle en el que se realizan los cálculos incluye una llamada al kernel de GPU (Cálculo de resultados parciales), seguida de una transferencia de los resultados de la GPU al Host, y por último una llamada a la función de host (Normalización del resultado).

Se hacen los siguientes cambios respecto a la versión inicial:

1. Se declara una tarea englobando la transferencia de memoria y la llamada a la función de host (en adelante, "tarea de host").

2. Se incluye una cláusula taskwait después de la llamada al kernel.

Estos cambios permiten que la tarea que engloba la transferencia de memoria y la función de host se ejecute en paralelo con el kernel de la siguiente iteración. Con esta versión, las ejecuciones más rápidas (menor número de iteraciones y matrices más pequeñas), tardan menos en la versión de referencia, siendo los tiempos de OmpSs de aproximadamente el doble. Sin embargo al aumentar tamaño de entrada y número de iteraciones se consiguen tiempos mejores, obteniéndose mayores mejoras respecto a la referencia cuanto mayor es la carga de trabajo del programa, entre un 4% y un 20%.

Es probable que sea inevitable obtener tiempos considerablemente mayores con OmpSs en ejecuciones muy rápidas, ya que en estos casos el overhead debido al planificador de tareas es mucho más relevante.

5.1.3 Primera versión con dependencias

Se hace una nueva versión de la aplicación con el objetivo de utilizar el sistema de dependencias de OmpSs para controlar el orden de ejecución de las tareas, en lugar de puntos de sincronización explícitos.

Los cambios respecto a la versión anterior son los siguientes:

- Se declaran en el kernel dependencias de entrada y salida por las matrices correspondientes en la memoria del dispositivo.

- Se declara en la tarea de host una dependencia de entrada por la matriz de salida del kernel, en la memoria del dispositivo.

- Se declara la tarea de host como región de exclusión mútua usando la cláusula commutative de OmpSs sobre la matriz auxiliar que utiliza la función de host.

Esta cláusula commutative es necesaria debido a que el bucle distingue entre iteraciones pares e impares, y cada una utiliza matrices diferentes (La que en una iteración se toma como entrada, en la otra es la salida, y viceversa). Esto provoca que no se establezcan dependencias entre la tarea de host de una iteración y la de la siguiente, lo que puede llevar a que ambas se ejecuten al mismo tiempo, sobreescribiendo una la matriz auxiliar de la otra. Con esta versión, los tiempos obtenidos están en el rango entre la versión de referencia y la versión naive.

5.1.4 Segunda versión con dependencias

Se usan los profilers de CUDA (nvprof) y Ompss (extrae) para determinar posibles causas para las diferencias en el tiempo de ejecución observadas en aplicaciones que en teoría deberían tener un comportamiento idéntico.

Se utiliza como caso de prueba los argumentos de entrada 10000, 10: una matriz de tamaño 10000 $\times$ 10000 elevada a 10. Con estos argumentos, la ejecución de la versión naive tarda unos 18.5 segundos, mientras que la de la versión con dependencias termina en unos 22 segundos.

Las diferencias en los tiempos se deben a diferencias en el tiempo entre la finalización de un kernel y el lanzamiento del siguiente, es decir, a diferencias en el tiempo de ejecución de la tarea de host.

Estos tiempos son, invariablemente, de aproximadamente un segundo en el caso de la versión naive. En la versión que utiliza dependencias, aparecen intervalos de un segundo, pero en ocasiones el tiempo aumenta hasta alrededor de 1.75 segundos.

La causa de esos aumentos es que, al acabar un kernel, se satisfacen las dependencias tanto de la tarea de host como del kernel de la siguiente iteración. Si el scheduler ejecuta primero la tarea de host, estamos en el caso equivalente a la versión naive. Sin embargo, si se ejecuta primero el kernel, aunque la tarea de host se lanza inmediatamente después, la copia de memoria va al stream por defecto de la GPU, y esto provoca que se quede bloqueada hasta que el kernel termina. Por tanto, al tiempo de ejecución de la tarea de host se suma el tiempo de ejecución del kernel, en este caso, unos 0.7 segundos.

El motivo de que no ocurra lo mismo en la versión naive es que en esa versión tenemos un taskwait después de cada kernel, lo que provoca que, en cada iteración, entren al scheduler primero la tarea de host, y después el kernel, de forma que siempre se lanza la copia antes que el kernel. En la versión con dependencias, sin embargo, ambas funciones están en la cola del planificador en el momento en que se liberan las dependencias.

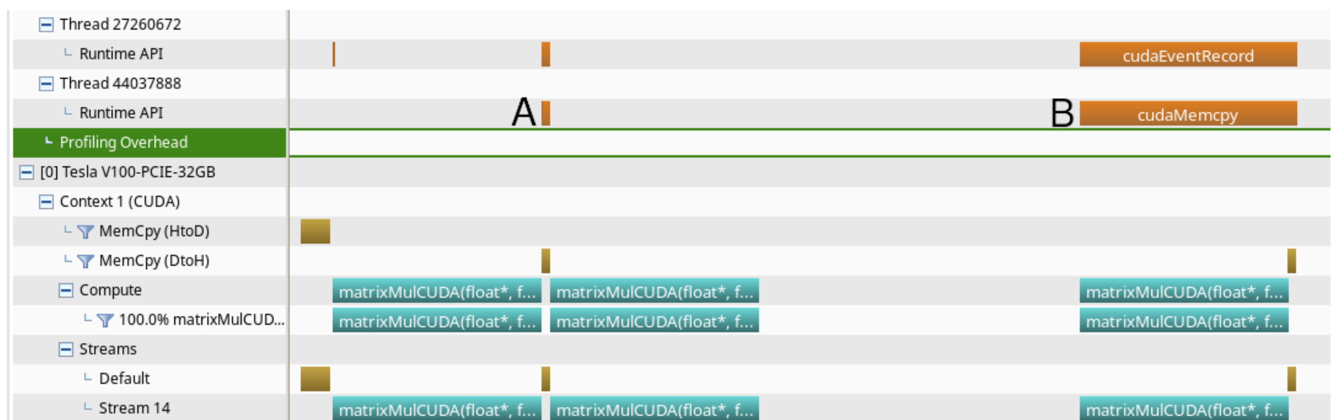

Figura 5.1: Resultados en el profiler para la version de OmpSs. La **A** indica una transferencia de memoria que se comporta de la forma esperada, ejecutándose antes que el kernel, la **B** indica una transferencia de memoria que se ejecuta después del kernel.

En la figura 5.1, se puede observar que la transferencia de memoria indicada por la **A** comienza antes que el kernel (Indicado en azul). El hilo de CPU correspondiente, por tanto, sólo tiene que esperar a que termine esta copia de memoria.

En el caso de la copia **B**, al comenzar después que el kernel y estar asignada al stream por defecto, esta no se realiza hasta que el kernel termina. El hilo de CPU se queda bloqueado en la copia y no llama a la función de normalización, que podría ejecutarse a la vez que el kernel si la copia se hubiera hecho antes.

En vista a las conclusiones obtenidas se hacen los siguientes cambios:

- Se divide la tarea de host en dos, una tarea con la llamada a la transferencia de memoria de GPU a Host, y otra con la llamada a la función de Host.
- Se declara en la tarea con la copia de memoria una dependencia de entrada por la matriz de salida del kernel, en memoria del dispositivo. Se declara también una dependencia de salida por la matriz de destino de la transferencia, en memoria del Host.
- Se declara una dependencia de entrada por la matriz de destino de la transferencia de memoria en la tarea con la llamada a la función de host.

En esta versión ya no existe el conflicto que aparece anteriormente. Las transferencias de memoria ya no están ligadas a la función de host, y por tanto no necesitan ejecutarse en exclusión mútua con la función de host de la siguiente iteración. De esta forma, las transferencias en las primeras iteraciones pueden

ejecutarse al acabar el kernel correspondiente (En la figura 5.1, la transferencia **B** podría ejecutarse inmediatamente después del kernel si no tuviera que esperar a que terminara la función de host).

Las funciones de host tardan mucho en ejecutarse en relación a los kernels y las transferencias, y a partir de cierto punto las transferencias tienen que esperar a que estas acaben. Esta situación es similar a lo que ocurre en la versión anterior. No hay problema, sin embargo, porque aunque ocurriera que una transferencia tuviera que esperar a que acabe un kernel, los datos siempre estarán disponibles en el Host antes de que sean necesarios.

Figura 5.2: Resultados en el profiler para la nueva versión. Las letras mayúsculas indican los kernels, las minúsculas indican las transferencias de los resultados de cada kernel.

En la figura 5.2 se muestran las primeras iteraciones con esta versión de la aplicación. Inicialmente se ejecutan 4 kernels sin esperas entre sí. Los dos primeros se ejecutan uno a continuación del otro ya que las matrices de destino en memoria de dispositivo están libres para ambos. El tercero y el cuarto se encuentran en la misma situación una vez se transfieren los resultados de los dos primeros al host.

A partir de ahí, es necesario esperar a que termine la función de host de **A** para realizar la siguiente transferencia, y poder ejecutar el siguiente kernel. La transferencia **c**, en este caso, se ha ejecutado antes que el kernel, sin embargo, aunque esto no ocurriera así no causaría un retraso en la ejecución, ya que la función de host asociada a **C** no puede comenzar hasta que termina la de **B**. Esto ocurre cuando comienza la transferencia **d**. Por tanto, el tiempo de ejecución en esta versión depende casi exclusivamente del tiempo de ejecución de la normalización de la matriz.

Con esta nueva versión se obtienen tiempos similares a la versión naive para las ejecuciones más cortas (tarda apoximadamente el doble que en la referencia). Los tiempos mejoran en el resto de ejecuciones respecto a la versión naive, con mayores mejoras cuanto mayor es la carga de trabajo, desde alrededor de un 3% hasta casi un 17%.

5.1.5 Tercera versión con dependencias

Se observa que, al ejecutarse las transferencias de memoria de la GPU en el stream por defecto, se está perdiendo la oportunidad de solapar su ejecución con los kernels.

Se declaran dos nuevos streams de CUDA en la aplicación, y se cambian las llamadas a las transferencias de memoria por transferencias asíncronas, que admiten la ejecución en streams concretos.

Se incluye una llamada de sincronización del stream utilizado en las transferencias de memoria después de la llamada a las propias transferencias. Esto es necesario en Ompss: Al tratarse de una llamada asíncrona, el hilo de host vuelve inmediatamente, por lo que, si no se hace que espere, el planificador daría la tarea por terminada y liberaría sus dependencias antes de tiempo.

No se espera una gran mejora para esta modificación, ya que las transferencias de memoria son la operación que menos tarda en el programa. Sin embargo, en ejecuciones con matrices grandes, en las que las transferencias son más relevantes, se obtienen mejoras respecto a lo anterior. En los casos de prueba con matrices de tamaños 6144 y 8192 la mejora es de alrededor de un 4%

5.1.6 Error de memoria

Haciendo pruebas con diferentes directivas en matrixpow, se llega a un código en el que se utiliza la directiva task for para paralelizar un bucle. Este código falla en la mayoría de ejecuciones con un error diciendo que no se puede reservar suficiente memoria.

Se hace un programa para reproducir el error a partir de matrixpow. Este programa toma los mismos argumentos que el original. Reserva memoria en el host utilizando el tamaño indicado en los argumentos, y pasa a ejecutar un bucle con tantas iteraciones como se indique.

Dentro de este bucle se ejecuta un bucle vacío con tantas iteraciones como elementos tiene la matriz reservada. Este bucle se paraleliza utilizando un task for, al que se le asigna una dependencia inout por la matriz.

Se ejecuta una batería de pruebas para determinar la causa del error. En estas pruebas, se ejecuta el programa con tamaños de la matriz 10×10, 100×100, 1000×1000 y 10000×10000. Para cada uno de esos tamaños, se prueban 10, 100, 1000 y 10000 iteraciones del bucle exterior. Por último, para cada número de iteraciones, se hacen ejecuciones variando el número de hilos desde 1 hasta 96. Estos tamaños, iteraciones, y números de hilos se eligen así para tener un rango amplio de valores en cada una de las variables que se cree que intervienen a la hora de reproducir el error.

Se determina que los factores principales para que se produzca el error son el número de iteraciones del bucle y el número de hilos disponibles para el programa. La cantidad de memoria a la que afecta la dependencia parece tener un efecto mucho menor. Por ejemplo, en la ejecución con 1000 iteraciones el primer fallo para una matriz de 10000×10000 se dió con 36 hilos, para una de 1000×1000, con 41, para la de 100×100, con 42, y por último, para la de 10×10, con 43.

Se pregunta al soporte técnico de OmpSs-2, en el Centro de Supercomputación de Barcelona, sobre este error. El fallo está asociado a OmpSs-2@cluster, y se produce cuando no se puede reservar memoria para crear nuevas tareas. OmpSs-2@cluster es la versión del modelo que incluye funcionalidades para el reparto de memoria y tareas entre varias máquinas. Es la versión que se instala para el proyecto en vista a poder ejecutar en un futuro aplicaciones de OmpSs en varios nodos del cluster.

Para solucionarlo, se sugiriere que se aumente la cantidad de memoria disponible para el allocator de OmpSs utilizando ulimit. También, que se modifique la variable de entorno NANOS6_SPACE_GLOBAL_ALLOC_SIZE, que determina el límite de memoria para el allocator.

Ulimit permite establecer límites para el uso de recursos de los programas que lance la shell desde donde se ejecuta. En este caso, eliminar el límite para el tamaño del stack hace que haya que aumentar el número de iteraciones hasta que el programa comienza a fallar de nuevo. Por otro lado, modificar la variable de entorno parece no tener ningún efecto.

5.2 Hotspot

5.2.1 Primeros casos de prueba

A partir del código de referencia, se llega a la conclusión de que la altura de la pirámide debe ser menor que el número total de iteraciones, ya que a la variable que controla el bucle principal se le suma, en cada iteración, la altura de la pirámide. Asimismo, el número de iteraciones por copia debe ser también menor que el total. De lo contrario, nunca se ejecutaría el código de host.

Basándose en esto se eligen una serie de casos de prueba seleccionados en función del tiempo que tarda en ejecutarse el código nativo con los mismos. Se seleccionan cuatro casos de prueba, con tiempos de ejecución en el código nativo desde unos 0.23 segundos hasta cerca de 10 segundos. Más adelante, se pasa a usar como casos de prueba las mismas entradas usadas para hacer pruebas en la versión de Controller.

5.2.2 Versión inicial

Inicialmente se modifica el código de referencia de la misma forma que matrixpow, lanzando el kernel como una tarea de OmpSs. Los tiempos de ejecución observados en esta versión inicial son muy similares a los de referencia, aunque algo superiores, del orden de entre un 2 y un 7% más en función del caso de prueba, y con mucha variabilidad entre ejecuciones. En el caso con menor carga computacional sí se observan tiempos mucho peores para esta versión, con ejecuciones cerca de un 50% más largas. Es probable que esta diferencia se deba al sobrecoste introducido por el planificador de tareas.

5.2.3 Primera versión con dependencias

Se hacen una serie de cambios para que la parte de host se ejecute en paralelo con los kernels de GPU:

- Se declara una tarea que contenía la transferencia de memoria de GPU al host, y la llamada a la función de host (en adelante, "tarea de host").
- Se declara una dependencia de entrada en la nueva tarea por la matriz de salida del kernel, en memoria de dispositivo.
- Se declara la funcion de host como región de exclusión mútua por la matriz de destino en memoria del Host, utilizando la cláusula commutative.
- Se declara una dependencia de entrada en el kernel por la matriz de origen, en memoria de dispositivo, y otra de salida por la matriz de destino, también en memoria del dispositivo.

En cada iteración del bucle principal se intercambian los índices que determinan la matriz de origen y de destino. De forma similar a lo que ocurre en matrixpow, esto lleva a que no se establezcan dependencias entre las tareas de host de iteraciones consecutivas.

Esto no es un problema ya que en iteraciones consecutivas las matrices donde se almacenan los datos al hacer la transferencia de memoria son diferentes. Sin embargo es necesario que las tareas que actúan sobre la misma matriz se ejecuten en exclusión mútua. La función de la cláusula commutative es que las tareas de host de las iteraciones pares no se ejecuten simultáneamente, y lo mismo con las iteraciones impares.

Con los casos de prueba iniciales, que hacen el copyback cada 50 iteraciones, los tiempos de esta versión son alrededor de un 10% mayores a los de la referencia.

Cuando se usan los mismos casos de prueba que para el programa de Controllers, en los que se hace el copyback todas las iteraciones, se obtienen tiempos de ejecución menores para la versión de OmpSs en comparación a la de referencia. Entre un 10% más rápidos, en la ejecución con menor número de iteraciones, y alrededor un 40%, para el resto.

Esta diferencia se debe a que la ventaja que se obtiene con la aplicación en OmpSs es que la sección de host se puede solapar con la ejecución del kernel en GPU. Sin embargo, sólo se están solapando operaciones en las iteraciones en las que se ejecuta la parte de Host. En los casos de prueba que se eligien inicialmente, la tarea de Host se ejecuta en muy pocas ocasiones, por lo que la ejecución es secuencial la mayor parte del tiempo.

5.2.4 Segunda versión con dependencias

Se detecta un error en la aplicación descrita en la versión anterior: Aunque no sea necesaria la exclusión mútua en la ejecución de la tarea de Host en iteraciones consecutivas, sí es necesaria para la ejecución de la función de Host, que realiza una operación memcpy, ya que esta opera sobre el mismo buffer en todas las iteraciones. Este problema no se detecta anteriormente debido a que la tarea de host no influye en la salida del programa.

Se realiza la siguiente modificación para solucionarlo:

- Declaración de una subtarea dentro de la tarea de Host, que contiene la llamada a la función host_compute. Esta subtarea declara una región de exclusión mutua utilizando la cláusula commutative sobre el buffer de memoria que utiliza la función.

Se declara como una subtarea ya que la función de host tiene que seguir estando dentro de la región de exclusión mutua por la matriz de destino de la transferencia.

Como era de esperar, este cambio hace más lentas las ejecuciones de la aplicación, aunque se siguen obteniendo mejoras respecto a la referencia. En el caso de prueba con menos iteraciones, la mejora sigue estando cerca de un 10%, en el resto, oscila entre un 30 y un 40%.

5.2.5 Versión con solapamiento en GPU

El profiler muestra que, aunque en la versión anterior sí se consigue solapar la ejecución de la operación memcpy con los kernels, no ocurre lo mismo con las transferencias de memoria. Al ejecutarse estas en el stream por defecto de la GPU, no pueden ejecutarse en paralelo con los kernels.

Se hacen los siguientes cambios para conseguir el solapamiento de estas operaciones:

- Creación de dos streams de CUDA durante la fase de inicialización de la aplicación.
- Se cambian las llamadas a transferencias de memoria síncronas por otras asíncronas, que permiten la ejecución en streams concretos. Se utilizan streams diferentes en iteraciones consecutivas.
- Se introduce una llamada a CudaStreamSynchronize para que el hilo que ejecuta la tarea de host espere a la finalización de la transferencia.

Con esta versión, se obtienen pequeñas mejoras en cuanto al tiempo de ejecución respecto al caso sin solapamiento. La mayor mejora la vemos en la ejecución con menor carga de trabajo, siendo alrededor de un 9% más rápida, y un 25% más que la referencia. En el resto de casos vemos mejoras desde un 1 a un 7%. Hay mucha variabilidad entre ejecuciones, pero por lo general esta nueva versión siempre es más rápida que la que no solapa operaciones.

5.2.6 Tercera versión con dependencias

Se hace una modificación en el programa para poder detectar errores en los resultados intermedios: Calcular e imprimir la norma del resultado de cada iteración. El propósito de esta modificación es comprobar que los resultados sigan siendo correctos después de cada modificación a la aplicación, pero no se usa en las ejecuciones para determinar la eficiencia del programa.

Esta comprobación revela que los resultados intermedios de la versión de OmpSs no son los mismos que los de la referencia. En algunas ocasiones estos resultados aparecen símplemente desordenados, mientras que en otras son totalmente diferentes.

Se detectan dos problemas: El primero, que la tarea de host no espera a que termine su tarea hija, la llamada a host_compute, antes de terminar y liberar sus dependencias. Esto es el comportamiento por defecto de OmpSs. En este caso, esto lleva a que la llamada a host_compute no esté dentro de la región de exclusión mútua por la matriz de destino de los resultados, es decir, permite que, en ocasiones, se actualice la matriz de destino antes de que host_compute pueda almacenar los resultados, o que se actualice mientras lo está haciendo.

Esto nos lleva al segundo problema: Aunque se consiga que la llamada a host_compute se ejecute en exclusión mútua por la matriz de destino de los resultados, por ejemplo con una cláusula taskwait detrás de la llamada, sigue siendo posible que, en un momento dado, aparezcan en el scheduler las llamadas a host_compute de dos iteraciones consecutivas. Ante esta situación, el scheduler elige una de las tareas. Si elige la tarea de la segunda iteración, los resultados aparecen en orden incorrecto: la norma de la iteración i+1 antes que la de la iteración i.

Una de las soluciones que se prueban es incluir una cláusula taskwait después de la llamada a host_compute, en respuesta al primer problema, y establecer la prioridad de las llamadas a host_compute en función de la iteración. Con esta versión de la aplicación, los resultados son mucho más cercanos a los de referencia, pero ocasionalmente siguen apareciendo en orden incorrecto. El motivo de estos errores es que, de forma muy puntual, la transferencia de memoria de la primera iteración termina más tarde que la de la segunda. Si se da esta situación, la llamada a host_compute de la segunda iteración entra al scheduler y, al no haber otras tareas esperando, pasa a ejecutarse.

Se llega la siguiente versión de la aplicación en la que no aparece este error:

- Se separa la tarea que contiene a host_compute de la tarea de host.
- Se declara una dependencia de salida en la tarea que ahora sólo contiene la transferencia de memoria por la matriz de destino de los resultados.

En este código, no hay que esperar a que termine la transferencia de memoria para que se envíe la llamada a host_compute al scheduler, lo que garantiza que siempre entrará antes la llamada de la primera iteración, de forma que, aunque se diera el caso de que la transferencia de memoria de la primera iteración acabe después de la segunda, las prioridades impedirían que la segunda llamada a host_compute se ejecutase primero.

También se observó que, aunque se eliminen las prioridades, los resultados siguen siendo correctos. Es complicado entender por qué sucede así, ya que, a priori, esto nos llevaría de vuelta a la situación en la que se podría ejecutar la segunda tarea antes que la primera, si ambas están esperando y sus dependencias se satisfacen.

Tal y como se esperaba, los tiempos de ejecución obtenidos en este caso son más lentos que en la versión anterior. En esta versión final, vemos una mejora de un 23% respecto a la referencia en la ejecución con menor carga de trabajo, que aumenta progresivamente hasta alrededor de un 30% en las ejecuciones más largas.

5.3 Sobel

5.3.1 Versión inicial

Se realiza toda la paralelización de la aplicación a partir de la versión de referencia que lee y almacena los fotogramas en ficheros, ya que la sección en la que se realizan los cálculos es esencialmente la misma en las cuatro versiones.

Todas las pruebas de la aplicación se realizan utilizando el mismo vídeo, por lo que la variación entre casos de prueba es el número de fotogramas procesados, cinco casos desde 100 hasta 500.

Se hizo, de nuevo, la misma adaptación que en los otros programas para lanzar el kernel como una tarea. Los tiempos de ejecución son muy similares a los de la versión de referencia (Se observan tiempos en torno a un 0.05% mayores). Por otro lado, las salidas producidas por el programa no coinciden con las de referencia.
Siendo la salida de esta aplicación ficheros con los fotogramas modificados, y no teniendo una forma de visualizar esta salida, se decide seguir con la paralelización tomando como referencia los resultados de esta primera versión, en vista de que los tiempos de ejecución son muy similares, el tamaño de los ficheros de salida idéntico, y que, en las otras dos aplicaciones, los resultados varían respecto a la referencia en pequeños porcentajes.

5.3.2 Primera versión con dependencias

Una de las principales características de la aplicación es que se procesan por separado cada uno de los tres componentes de un fotograma, de forma independiente.

En esta primera versión:

- Se declara la función de carga de fotogramas como una tarea de OmpSs.
- Se declaran tres dependencias de salida en la función de carga de fotogramas, una sobre cada matriz de entrada en el Host.
- Se declara la función de escritura de fotogramas como una tarea de OmpSs.
- Se declaran tres dependencias de entrada en la función de escritura de fotogramas, una sobre cada matriz de salida en el Host.
- Se declara una tarea que contenía la transferencia de Host a GPU, la llamada al kernel, y la transferencia de GPU a Host. En ejecución esto resulta en tres tareas, cada una de las cuales procesa un componente.
- Se declara para esa tarea una dependencia de entrada sobre la matriz de entrada correspondiente, en memoria de Host, y otra de salida sobre la matriz de salida correspondiente, también en memoria de Host.

Esta aproximación falla, ya que es necesario disponer de un mecanismo para liberar las dependencias en las funciones de host a medida que se van cargando y almacenando componentes y resultados. En OmpSs, este mecanismo es la cláusula release, que se usa para indicar que una tarea ha terminado los accesos conflictivos sobre una de las dependencias que aparecen en su declaración.

Si no se liberan las dependencias sobre cada componente por separado, el resultado es que la aplicación se ejecuta de forma secuencial, debido a las dependencias entre tareas. No se llega a obtener un programa en el que la cláusula release funcione correctamente.

5.3.3 Segunda versión con dependencias

Se realizan los siguientes cambios:

- Se eliminan las tareas que contienen a las funciones de Host.
- Se declaran nuevas tareas sobre las operaciones fread y fwrite, dentro de las funciones de host. De esta forma, en ejecución se crean tres tareas, una por componente.
- Se declara una dependencia de salida en la tarea de carga, por la matriz de entrada correspondiente en memoria de host.
- Se declara una dependencia de entrada en la tarea de escritura, por la matriz de salida correspondiente en memoria de host.
- Se elimina la tarea que contenía las llamadas a funciones de CUDA.
- Se declara la transferencia de Host a GPU como tarea.
- Se declara en esa tarea una dependencia de entrada por la matriz de entrada correspondiente en memoria de host, y otra de salida por la matriz de entrada correspondiente en memoria de dispositivo.
- Se declara en el kernel una dependencia de entrada por la matriz de entrada correspondiente en memoria de dispositivo, y otra de salida por la matriz de salida correspondiente, también en memoria de dispositivo.
- Se declara la transferencia de GPU a Host como tarea.
- Se declara en esa tarea una dependencia de entrada por la matriz de salida correspondiente en memoria de dispositivo, y otra de salida por la matriz de salida correspondiente en memoria de host.

En este punto, se están declarando las dependencias de forma incorrecta en el host, lo que provoca que se establezcan correctamente las dependencias entre tareas de host, pero no entre estas y el kernel. La solución para que se detecten es declarar una tarea que incluye sólo la llamada al kernel, y declarar en esa tarea las mismas dependencias que en el kernel. Así, se establecen dependencias entre el resto de tareas de host, y esa tarea de envoltorio.

Utilizando la función de OmpSs para generar una representación visual del grafo de tareas, se comprueba que las dependencias se establecen correctamente y el orden de ejecución es el esperado.

Los tiempos de ejecución obtenidos con esta versión del programa son más lentos que los de la referencia, a pesar de que, a priori, se estarían solapando correctamente las operaciones de carga y escritura de forogramas con los kernels.

5.3.4 Tercera versión con dependencias

Se determina, usando los profilers de CUDA y OmpSs, que el planificador de tareas de OmpSs está liberando las dependencias de la función de envoltorio antes de que el kernel termine. Esto ocurre así porque el hilo de host que llama a un kernel vuelve inmediatamente de la llamada. Por tanto, el planificador da la función de host por terminada.

En OmpSs, cuando una tarea termina, libera las dependencias que no hayan declarado también sus tareas hijas. En este caso, ocurre eso al ser incorrecta la declaración de las dependencias en el host, de forma que las dependencias declaradas en la tarea de envoltorio no son las mismas que las declaradas en el kernel.

En este momento, previo a encontrar el error en la declaración de las dependencias, se soluciona el error incluyendo una directiva taskwait después de la llamada al kernel, en la función de envoltorio en el host. Ese taskwait interrumpe la ejecución de la tarea hasta que el kernel termina, impidiendo la liberación de las dependencias antes de tiempo.

En esta versión, también se pasa a usar de nuevo las tareas que contenían las funciones de host, declaradas de la misma forma que en la primera versión con dependencias, ya que se obtienen mejores tiempos de esta forma. A priori parece más eficiente que el procesado de cada componente sea totalmente independiente de los otros dos, evitando una sincronización antes de cargar y almacenar datos. Sin embargo, es posible que el peor rendimiento que se observa se deba a que, de esta forma, se hacen peticiones intercaladas de lectura y escritura en disco, en direcciones diferentes.

Con esta versión del programa, en concreto tras incluir el taskwait después del kernel, las salidas obtenidas son iguales a las que produce la versión de referencia. Se obtienen tiempos mejores que los de la versión de referencia, entre un 12%, en la ejecución con 100 frames, y un 25% en la ejecución con 500 frames.

5.3.5 Cuarta versión con dependencias

Cuando se encuentra el error en la declaración de las dependencias, se modifica el programa para declararlas de forma correcta en las funciones de host, lo que permite eliminar tanto la función de envoltorio como la directiva taskwait para esperar a la finalización del kernel.
Esta versión obtiene tiempos menores a la anterior con todos los casos de prueba. La mejora respecto a los tiempos de referencia pasa a estar entre un 20 y un 25% (El tiempo mejora más en las ejecuciones cortas)

5.3.6 Versión con solapamiento en GPU

Se plantea crear una versión de la aplicación en la que cada transferencia de memoria vaya a un stream diferente, de forma que éstas no se interfieran entre sí ni con los kernels. En este caso, en vez de incluir llamadas a cudaStreamSynchronize, como en los otros programas, se usan eventos de CUDA para este mismo propósito. Se hacen los siguientes cambios:

- Se declaran e inicializan 6 streams de CUDA.

- Se declaran e inicializan 6 eventos de CUDA.
- Se declaran 2 arrays de 3 streams cada uno.
- Se declaran 2 arrays de 3 eventos cada uno.
- Se sustituyen las transferencias de memoria por sus versiones asícronas. Cada transferencia usa el stream que le corresponde, en función del componente y la dirección de la transferencia (De host a GPU o al revés).
- Se incluyen llamadas a cudaEventRecord y cudaStreamSynchronize tras las llamadas a las transferencias de memoria.

Los resultados son variados en este caso, y dependen de la versión de Sobel que se esté utilizando, y de la máquina. En las pruebas que se hacen en Manticore, las versiones con escritura en fichero empeoran alrededor de un 25% respecto a la versión sin streams. Por otro lado, las versiones con escritura en fichero mejoran un 9%, en el caso de lectura y escritura en memoria, y un 4%, si se leía de fichero.
Por otro lado, las mismas pruebas en pegaso dan mejoras de entre un 7 y un 10% para las versiones con escritura en fichero, las versiones con escritura en memoria también obtienen mejoras, de alrededor de un 20% en el caso de fichero a memoria, y de un 10% para la versión de memoria a memoria.

5.4 Cholesky

5.4.1 Generación de resultados de referencia

El programa original no proporciona ninguna salida que pueda ser utilizada para validar los resultados obtenidos con modificaciones de la aplicación. Para añadir esta funcionalidad al programa, se empieza por determinar dónde se almacenan los resultados de la aplicación.
Leyendo sobre el algoritmo, y comprobando la documentación de las funciones de LAPACK y BLAS utilizadas en el código original, se determina que los resultados se van almacenando en la mitad inferior de la matriz que se está procesando.

Se eligie, para hallar la norma de la matriz, la misma función utilizada en hotspot. Esta función calcula la suma de todos los elementos de la matriz al cuadrado, y después halla la raíz cuadrada de la suma.

5.4.2 Modificaciones a las estructuras internas del programa

En el programa original, se utiliza una matriz de cuatro dimensiones, dos para indexar submatrices, y dos para indexar dentro de esas submatrices. Para simplificar la transferencia y el uso de este array en GPU, se sustituye por un array unidimensional. Se crea una macro para acceder a este array, de forma que los accesos puedan seguir realizándose con los mismos índices. Se comprueba que los resultados siguen siendo los mismos después de este cambio.

Más adelante se hace necesario otro cambio a esta matriz. El código original inicializa la matriz en orden row-major, es decir, los elementos consecutivos de una fila de la matriz son contiguos en memoria. Sin embargo, internamente las librerías LAPACK y BLAS utilizan el orden column-major: Los elementos consecutivos de una columna son contíguos en memoria.

Las implementaciones de netlib de estas librerías que usa el código original permiten indicar en las llamadas a funciones si la matriz que se le pasa está en row-major, y hacen la transformación internamente para pasarla a column-major. Las implementaciones de MAGMA de estas mismas librerías (Las elegidas para la implementación en GPU del programa), no tienen esta opción, sólo admiten matrices en orden column-major.
Por tanto, se modifican los accesos a la matriz para que el orden de los datos en memoria fuera el adecuado. Tras este cambio, se observa que los resultados varían ligeramente respecto a los obtenidos con la matriz en row-major. Lo más probable es que esta diferencia se deba a cambios en el orden de las

operaciones.
También se observa que los tiempos de ejecucion son más rápidos al usar la matriz en column-major. Esto es lo esperable, ya que cuando se utilizan matrices row-major, las funciones tienen que trasponer la matriz internamente antes de poder usarla.

5.4.3 Primera versión con GPU

Se hace una primera versión del programa con operaciones en GPU a partir de la versión secuencial del BSC. Para ello, además de las modificaciones explicadas previamente:

- Se reserva suficiente memoria para la matriz en GPU.
- Se sustituye la llamada a la función "dtrsm" de LAPACK por su equivalente en la implementación de LAPACK de MAGMA.
- Se incluye una transferencia de memoria de la matriz del host a la de GPU justo antes de la llamada a esta función, y una transferencia de vuelta después.

Se comprueba que los resultados al incluir esta función siguen siendo iguales a los de la referencia con la matriz en column-major.

5.4.4 Versión secuencial con GPU

El siguiente paso es sustituir el resto de funciones, en este caso de BLAS, por sus equivalentes en MAGMA:

- Se sustituyen las funciones dtrsm, dgemm, y dsyrk, por sus equivalentes en la implementación de BLAS de MAGMA.
- Se incluye una transferencia de la matriz a GPU antes de la llamada a cholesky, y una de vuelta después de la llamada.

En este caso, los resultados obtenidos son diferentes tanto a los de la versión de CPU con la matriz en row-major, como a los de la versión con la matriz en column-major.
A priori, esto se debe a que las operaciones se realizan es distinto orden en GPU, y a que las GPUs de nvidia usan la operación FMA (Fused multiply-add [4]), que no está implementada en la arquitectura x86, y también afecta a los resultados.

En la mayoría de casos de prueba utilizados, los resultados de una y otra versión comparten un número razonable de cifras significativas. Sin embargo, se han encontrado entradas para las cuales los resultados difieren mucho. Siendo esta versión una traducción directa de las llamadas de la versión original por sus versiones de MAGMA, a priori estas diferencias sólo se podrían achacar a las diferencias en las operaciones de punto flotante previamente mencionadas, o a diferencias entre las implementaciones de las propias funciones.

Por último, los tiempos de ejecución obtenidos con esta versión del programa son alrededor de un 95% menores que los de su equivalente con las funciones de CPU.

5.4.5 Primeras versiones de Ompss

Se crean dos versiones de OmpSs a partir de la aplicación descrita en la sección anterior. Para ello, símplemente se trasladan las directivas de OmpSs de las versiones originales del BSC a esta nueva versión en GPU. El mayor cambio en este caso es sustituir las direcciones de memoria en las dependencias por su equivalente con el array plano y en GPU.

Los resultados obtenidos al trasladar las directivas de OmpSs de esta forma, sin embargo, son incorrectos, siendo completamente diferentes a los de referencia. Probando con sincronizaciones explícitas después de las tareas, se llega a que es necesario incluir un taskwait explícito después de la llamada a

dtrsm. Esto ocurre pese a que el grafo de dependencias es exactamente igual en esta versión que en las versiones originales.

En cuanto a tiempos de ejecución, aunque estas versiones siguen siendo más rápidas que sus equivalentes en GPU, son más lentas que la versión en GPU sin tareas. Esto se debe a que, aunque tengamos varios hilos en host, todas las operaciones van al mismo stream en GPU, por lo que se están ejecutando de forma secuencial.

5.4.6 Versión con streams

Se decide hacer una versión de la aplicación en la que se utilicen varios streams en GPU, permitiendo que las operaciones se ejecuten en paralelo.
Las funciones de BLAS de MAGMA admiten un argumento para indicar la cola del dispositivo en la que ejecutar esas operaciones. En el caso de CUDA, estas colas pueden crearse a partir de un stream de CUDA.
Para incluir esto en las versiones de OmpSs:

- Se declara un array de colas de MAGMA con tantos elementos como hilos de host tiene disponible el programa.
- Se inicializan esas colas de MAGMA con un array de streams de CUDA.
- Se declara una variable que indica el índice de la siguiente cola a utilizar.
- Se declara, dentro de cada tarea de OmpSs, una sección crítica en la que se toma la cola correspondiente para la operación de esa tarea, y se incrementa el índice de la cola que tiene que utilizar la siguiente tarea.
- Se incluye una directiva de sincronización de stream después de cada llamada a una función de GPU. Ya que estas funciones son asíncronas, esto es necesario para que OmpSs no de la tarea por terminada antes de tiempo.

Todas estas nuevas estructuras necesarias se declaran como variables globales, y se crea una función para inicializarlas, ya que si se inicializan dentro de la llamada a cholesky, ese tiempo se contabiliza como tiempo de ejecución.

Se comprueba que los resultados siguen siendo iguales a los obtenidos con las versiones anteriores. Los tiempos obtenidos en este caso son alrededor de un 85% más rápidos que sus contrapartidas en CPU.

5.5 Problemas de implementación y de despliegue

Se detectan una serie de diferencias en cuanto a tiempos de ejecución entre los ejecutables generados por el compilador de nvidia, nvcc, y el de OmpSs, Mercurium, donde en ocasiones los tiempos son muy distintos a pesar de tratarse de la misma aplicación. Se realizan una serie de pruebas con las versiones nativas en Cuda de matrixpow, hotspot y sobel para determinar posibles causas de las diferencias, y su posible impacto en los tiempos de las versiones paralelizadas con OmpSs.

Se utilizan los siguientes programas para las pruebas:

- Versión de CUDA síncrona de matrixpow
- Versión de CUDA asíncrona de matrixpow
- Versión de CUDA síncrona de hotspot
- Versión de CUDA asíncrona de hotspot
- Versión de CUDA síncrona de sobel (File to File)

- Versión de CUDA asíncrona de sobel (File to File)

Cada uno de ellos compilado con nvcc y Mercurium.
Como casos de prueba se utilizan las mismas entradas que las usadas para la experimentación, variando el número de hilos entre 1 y 96 y obteniendo la media de 5 ejecuciones del mismo caso de prueba para cada número de hilos.

Los resultados fueron los siguientes:

- Para las versiones síncronas, los tiempos de ejecución totales de los ejecutables compilados con Mercurium son menores que los obtenidos con nvvp. Sin embargo, los tiempos de ejecución de la sección donde se realizan los cálculos son los mismos.
- Con 1 y 2 hilos, la versión asíncrona de matrixpow tarda lo mismo con ambos compiladores, a partir de 3 hilos, la versión compilada con nvcc tarda aproximadamente la mitad que la de Mercurium.
- En hotspot, los resultados obtenidos con nvcc son cerca de un 35% más rápidos que los de Mercurium.
- En sobel, los tiempos de nvcc son alrededor de un 45% más rápidos con 1 sólo hilo. Esa cifra desciende hasta alrededor de un 20% con 96 hilos.

A continuación se muestra un conjunto reducido de los resultados obtenidos. Mostrar todas las entradas de cada programa parece excesivo, dado que el comportamiento es muy similar en todas ellas.

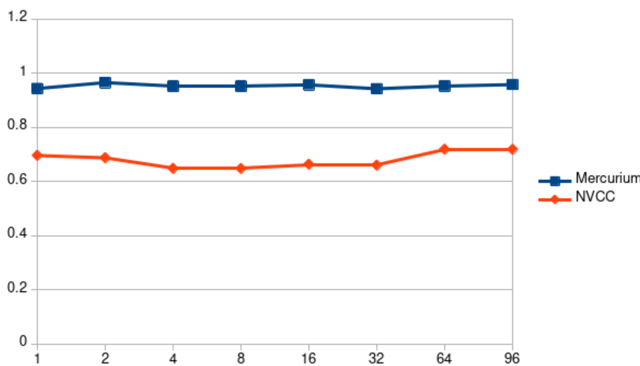

Figura 5.3: Hotspot con entrada 2500 4 500 1

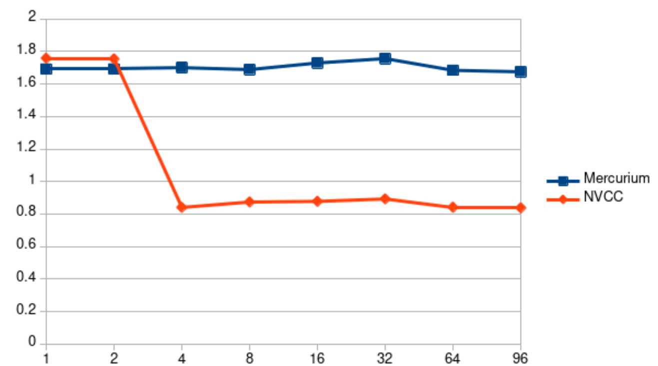

Figura 5.4: Matrixpow con entrada 1024 40 0

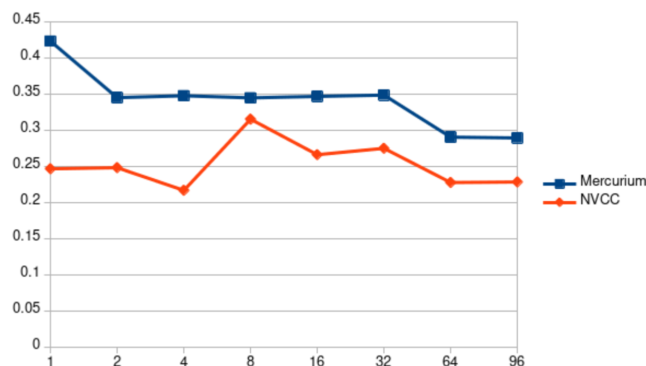

Figura 5.5: Sobel con entrada 100

No se ha conseguido explicar, en el caso de sobel y hotspot, el porqué de los resultados obtenidos. Especialmente preocupante es el caso de sobel, donde se observa una clara tendencia en los tiempos de la versión compilada con Mercurium según aumenta el número de hilos, a pesar de que esta es una versión nativa que no usa OmpSs y en principio no debería beneficiarse de un aumento en el número de hilos.

Sin embargo, sí se ha determinado por qué vemos estos resultados en matrixpow, y eso puede ser una pista para las otras aplicaciones.
En matrixpow, es interesante determinar por qué los tiempos de nvcc bajan a la mitad a partir de 3 hilos. A priori, el factor más importante es la máquina en la que se está ejecutando. Estas pruebas se realizaron en la máquina manticore, que usa procesadores de Intel con hyperthreading. Típicamente, una aplicación que use dos hilos en el mismo núcleo se ejecutará más lentamente que si usara hilos en núcleos separados.

Al usar llamadas asíncronas de CUDA, y viendo los resultados, parece seguro asumir que la aplicación compilada con nvcc está usando dos hilos, cuando el programa sólo dispone de un hilo físico, o de dos, pero en el mismo núcleo, los tiempos de ejecución son mayores. Al contrario, cuando se dispone de hilos físicos en más de un núcleo, el tiempo se reduce.

Para comprobar esta teoría se utiliza el paquete hwloc. Mediante esta herramienta, entre otras funcionalidades, es posible asignar explícitamente los recursos para la ejecución de una aplicación, cosa que no es posible con slurm, la herramienta usada en el cluster para asignar las ejecuciones a cada máquina.

Usando hwloc, se comprueba que si se asignan dos hilos a la aplicación, pero estos están en núcleos separados, el tiempo de ejecución es el mismo que el obtenido con más hilos. Sin embargo, en el caso de la aplicación de Mercurium, independientemente de los hilos físicos que se le asignen los tiempos no varían. Es posible que esto se deba a diferencias en cómo trata Mercurium estas directivas asíncronas de CUDA, o a interferencias entre la forma en que la aplicación gestiona los hilos y cómo lo hace CUDA.

Se realizan varios intentos con hwloc para determinar las causas de los tiempos de Mercurium, aunque sin éxito. Como ejemplo, para asegurar que el problema no estuviera en que la aplicación usara el nodo NUMA más lejano a las GPUs, se probaron todas las combinaciones de nodo y GPU en cada programa, pero no se observó ningún cambio. Asimismo, se hicieron pruebas con la funcionalidad de hwloc que permite seleccionar los hilos físicos más cercanos al bus PCI de la GPU que se está usando, de nuevo, sin cambios.

5.6 Conclusiones

Se ha realizado, para cada uno de los tres programas del grupo Trasgo, una paralelización con OmpSs-2 buscando el solapamiento de transferencias de datos y cómputo. Asimismo, se ha implementado una versión en CUDA de la factorización de Cholesky, y se ha realizado una experimentación limitada en uno de los problemas encontrados con el compilador de OmpSs.

A continuación, se detallan las conclusiones obtenidas tras la implementación de tres aplicaciones en OmpSs.

- Es necesario separar tanto el código del kernel como su declaración en sendos ficheros, y compilarlos por separado del código de host tras declarar el kernel como una tarea. Una vez hecho eso, se debe comprobar la exactitud de los resultados de la nueva versión.
- Es necesario declarar las dependencias de la misma forma en el host y en los kernels, y de que si se declaran dependencias en subtareas, sea sobre datos compartidos por la tarea padre.
- Siempre que se introduzcan transferencias de datos asíncronas, es imprescindible incluír una sincronización explícita de CUDA dentro de la tarea para evitar que se liberen las dependencias prematuramente.
- Se debe tener en cuenta el comportamiento del scheduler cuando aparecen varias tareas listas para ejecutarse en la cola.

Capítulo 6

Experimentación

En este capítulo se presentan los resultados experimentales en cuanto a tiempo de ejecución obtenidos con las aplicaciones en OmpSs, y se comparan con los de Controllers y las versiones nativas.

6.1 Objetivo de la experimentación

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

Se dispone, para cada aplicación, de cinco implementaciones diferentes para realizar la comparativa: Dos versiones de referencia desarrolladas con las herramientas nativas de CUDA, una con transferencias de memoria síncronas y otra con transferencias asíncronas, que se solapan con el cómputo, Dos versiones desarrolladas con Controllers, de nuevo con transferencias síncronas y asíncronas, y por último la versión de OmpSs, donde no existe esa distinción entre síncrono y asíncrono, ya que todas las operaciones que se gestionen usando tareas de OmpSs serán siempre asíncronas respecto a otras tareas.

6.2 Metodología de experimentación

En las tres aplicaciones, se mide el tiempo de ejecución en segundos a través de funciones de OpenMP. En cada caso, se dispone de dos tiempos, uno global, que incluye los tiempos de inicialización del programa, y otro de ejecución, que sólo contabiliza el tiempo donde se están realizando cálculos. Los tiempos que aparecen en este documento son los de ejecución, ya que son los más relevantes de cara a medir la eficiencia de la paralelización.

Los datos de las versiones asíncronas y de Controllers fueron proporcionados por el grupo Trasgo. Para cada versión del programa, se dispone de los tiempos de ejecución de 30 lanzamientos por caso de prueba. El tiempo mostrado para cada aplicación es la media de estos tiempos.

Para la experimentación con OmpSs, se creó un script de python que lanza 30 veces cada caso de prueba y obtiene la media de los tiempos de ejecución. Previo a la realización de las pruebas de cada programa en cada máquina, se ejecutó la aplicacion variando el número de hilos y se eligió el número con el que se obtenían mejores tiempos. Es de esperar que el número de hilos disponibles para el programa haga variar el tiempo de ejecución en las versiones paralelizadas, aunque se observan efectos de ejecución extraños donde aumentar demasiado el número de hilos lleva a peores tiempos.

6.2.1 Escenario de experimentación

La batería de pruebas se ejecutará en dos máquinas del cluster del grupo Trasgo, Manticore y Pegaso. A continuación se hace un resumen de las características de ambas.

- Manticore: Consta de dos nodos NUMA, cada uno con una CPU Intel Xeon Platinum 8160, y 128GB de memoria RAM. En total, la máquina tiene un total de 48 cores físicos y 96 hilos. Está equipada con dos GPUs NVIDIA Volta V100 y dos AMD RadeonPro WX9100.

- Pegaso: Tiene una CPU Intel i7 960, con 4 cores físicos y 8 hilos junto a 7GB de memoria RAM. En el apartado de aceleradores, consta de dos NVIDIA Titan Black.

6.3 Resultados

A continuación se presentan en gráficas los resultados obtenidos para Matrixpow y Hotspot. También se incluyen, en tablas, los porcentajes de cambio en relación a OmpSs del resto de implementaciones. En estas tablas, los valores resaltados en rojo indican tiempos más lentos en la implementación con OmpSs, y los resaltados en verde, más rápidos. En el caso de Sobel, al usar un único caso de prueba de 100 fotogramas procesados, los resultados se presentan únicamente en forma de tabla.

6.3.1 Matrixpow

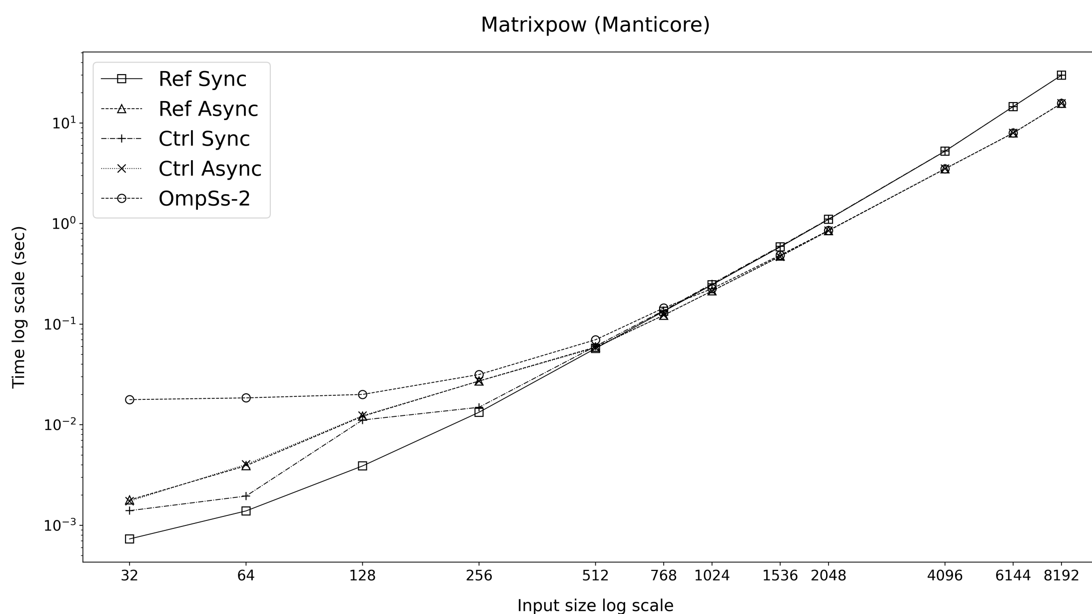

Figura 6.1: Tiempos de ejecución de Matrixpow en Manticore

Vemos que los tiempos obtenidos por OmpSs en las entradas más pequeñas son mucho mayores a los del resto de versiones. Esto era uno de los resultados que se esperaba obtener, y también se ha observado durante la implementación de las versiones de OmpSs. La causa más probable es el sobrecoste debido al planificador de tareas, que aumentaría el tiempo en una cantidad fija independientemente del tamaño de la entrada.

En cuanto a los casos con mayor carga computacional, vemos una clara mejora respecto a los tiempos obtenidos con la versión de referencia síncrona a partir de la que se ha realizado la implementación, y tiempos más lentos respecto a sendas versiones asíncronas. La diferencia con las versiones asíncronas, no obstante, disminuye claramente a medida que aumenta el tamaño de entrada.

En las ejecuciones en Pegaso, vemos un comportamiento de la aplicación diferente al de Manticore. Por un lado, OmpSs comienza a obtener mejoras respecto a las versiones síncronas con casos de prueba

Porcentaje de cambio en relación a OmpSs (Matrixpow + Manticore)

Size	Ref Sync	Ref Async	Ctrl Sync	Ctrl Async
32	-95.88%	-89.94%	-92.12%	-90.21%
64	-92.49%	-78.86%	-89.43%	-78.21%
128	-80.55%	-39.36%	-44.30%	-38.54%
256	-57.87%	-13.76%	-52.78%	-14.03%
512	-17.99%	-15.47%	-13.78%	-16.87%
768	-7.36%	-15.39%	-5.60%	-15.01%
1024	9.28%	-5.02%	11.78%	-5.20%
1536	20.67%	-2.27%	22.38%	-3.62%
2048	28.90%	-0.68%	29.76%	-0.54%
4096	49.73%	-0.08%	50.19%	0.15%
6144	82.99%	-0.15%	82.97%	-0.10%
8192	90.45%	-0.05%	90.29%	-0.14%

Tabla 6.1: Porcentajes de cambio respecto a OmpSs en Matrixpow + Manticore

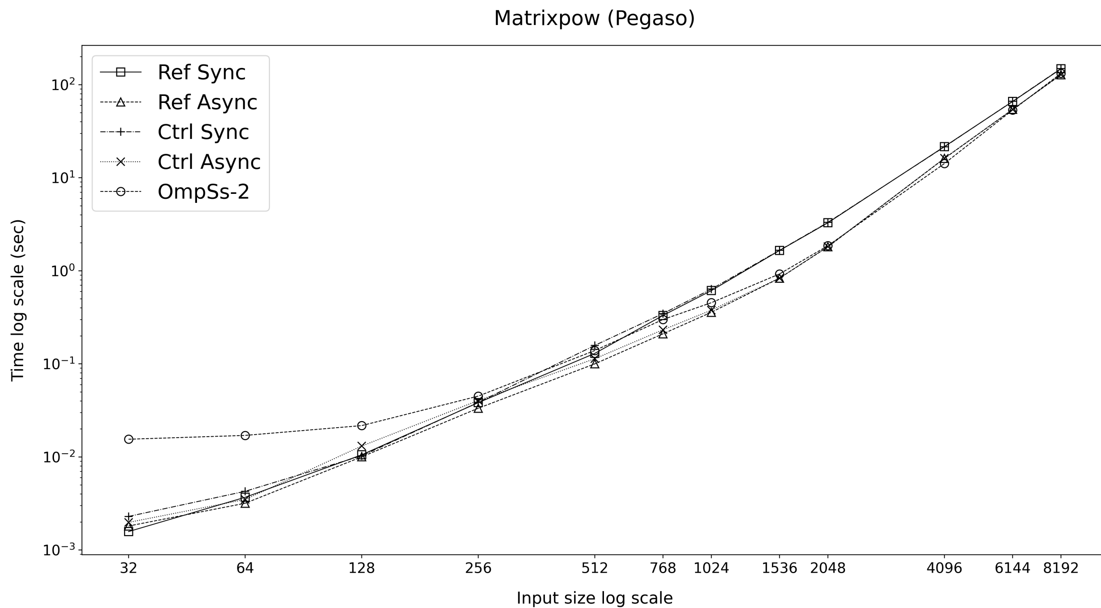

Figura 6.2: Tiempos de ejecución de Matrixpow en Pegaso

más pequeños. Además de esto, vemos cómo se obtienen mejoras respecto a las versiones asíncronas en dos de los casos de prueba.

Es posible que estas mejoras se deban a alguna característica concreta de esos casos de prueba en esta máquina, pero, como veremos más adelante, la versión de OmpSs de Hotspot se comporta también de forma similar en Pegaso. No se tiene una explicación definitiva al respecto, sin embargo, es posible que tiempos más largos de cómputo y transferencias de memoria ayuden a ocultar el sobrecoste causado por el planificador de tareas.

Porcentaje de cambio en relación a OmpSs (Matrixpow + Pegaso)

Size	Ref Sync	Ref Async	Ctrl Sync	Ctrl Async
32	-89.82%	-88.32%	-85.20%	-87.23%
64	-78.38%	-81.34%	-74.93%	-79.52%
128	-51.50%	-53.86%	-52.63%	-39.69%
256	-14.96%	-25.94%	-14.41%	-10.36%
512	-6.87%	-28.11%	13.50%	-18.33%
768	10.48%	-29.84%	15.77%	-22.44%
1024	34.91%	-21.14%	39.88%	-16.92%
1536	78.78%	-9.95%	79.71%	-9.87%
2048	76.43%	-2.69%	78.06%	-1.97%
4096	52.12%	13.46%	51.11%	14.11%
6144	23.46%	1.48%	24.09%	1.79%
8192	11.90%	-3.37%	12.66%	-2.82%

Tabla 6.2: Porcentajes de cambio respecto a OmpSs en Matrixpow + Pegaso

6.3.2 Hotspot

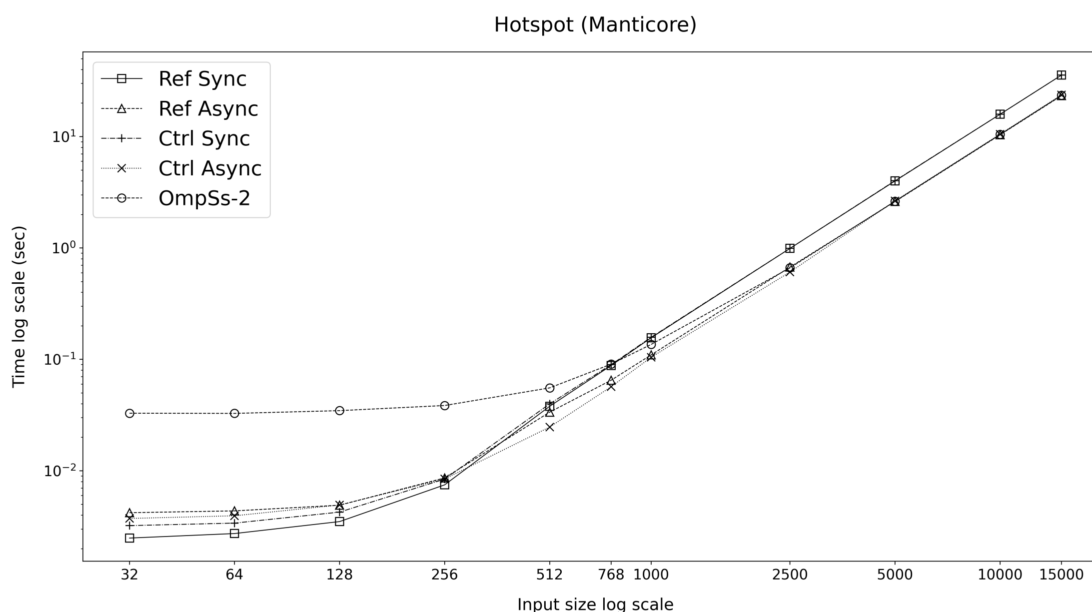

Figura 6.3: Tiempos de ejecución de Hotspot en Manticore

Los resultados en cuanto a las versiones síncronas son muy similares a los de Matrixpow en Manticore, obteniendo mejoras grandes en los casos de prueba con mayores entradas. Por otro lado, vemos cómo OmpSs obtiene resultados mejores que la versión asíncrona de Controllers para los tres casos de prueba con mayor carga computacional, y también que esta mejora aumenta ligeramente con el tamaño de la entrada.

Porcentaje de cambio en relación a OmpSs (Hotspot + Manticore)

Size	Ref Sync	Ref Async	Ctrl Sync	Ctrl Async
32	-92.42%	-87.20%	-90.20%	-88.64%
64	-91.65%	-86.66%	-89.64%	-87.95%
128	-89.90%	-85.83%	-87.72%	-85.77%
256	-80.60%	-77.62%	-78.14%	-78.09%
512	-31.65%	-39.44%	-28.16%	-55.50%
768	-2.71%	-28.25%	-1.07%	-37.28%
1000	14.68%	-19.48%	15.83%	-23.36%
2500	49.50%	1.62%	49.60%	-8.25%
5000	52.08%	-0.64%	52.86%	0.65%
10000	51.92%	-0.62%	51.97%	0.72%
15000	51.90%	-0.76%	51.86%	0.90%

Tabla 6.3: Porcentajes de cambio respecto a OmpSs en Hotspot + Manticore

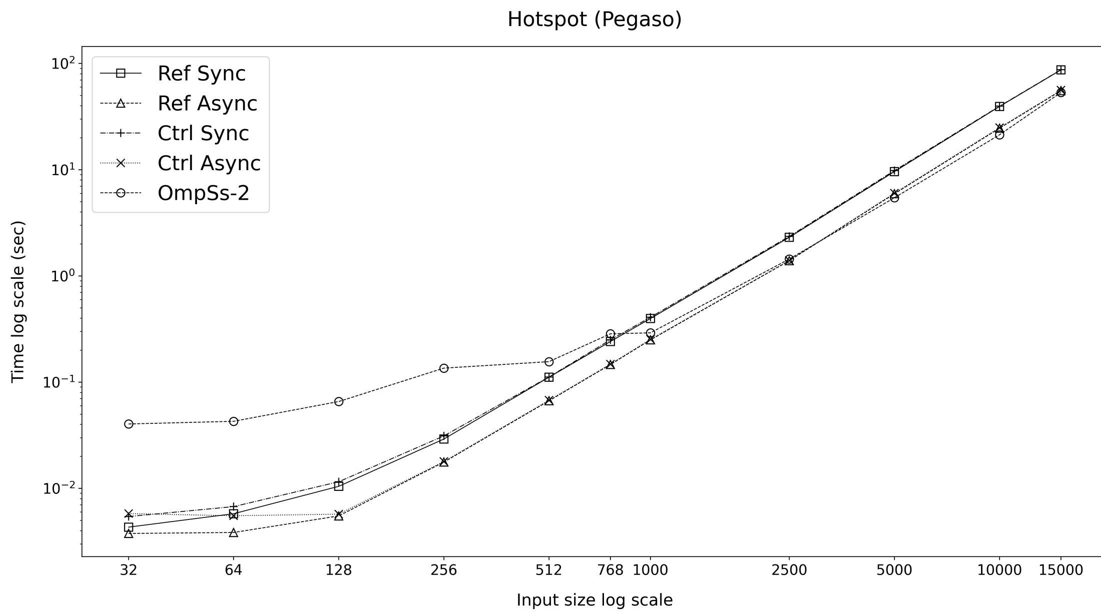

Figura 6.4: Tiempos de ejecución de Hotspot en Pegaso

El comportamiento respecto a las versiones síncronas es idéntico al de Manticore, aunque con mayores mejoras. En las asíncronas, OmpSs mejora tanto a la versión de referencia como a Controllers en los casos de prueba más largos. Los porcentajes de mejora respecto a Controllers son mayores que en Manticore, aunque en este caso no se puede ver una tendencia clara.

Porcentaje de cambio en relación a OmpSs (Hotspot + Pegaso)

Size	Ref Sync	Ref Async	Ctrl Sync	Ctrl Async
32	-89.32%	-90.68%	-86.56%	-85.66%
64	-86.51%	-91.02%	-84.27%	-87.11%
128	-84.09%	-91.62%	-82.43%	-91.29%
256	-78.51%	-86.96%	-77.02%	-86.81%
512	-28.66%	-57.23%	-28.23%	-56.68%
768	-15.38%	-48.66%	-12.27%	-48.16%
1000	36.38%	-14.18%	40.23%	-13.20%
2500	60.13%	-3.52%	63.34%	-3.10%
5000	76.31%	8.97%	80.05%	10.42%
10000	84.61%	15.25%	85.84%	16.63%
15000	63.10%	3.89%	63.90%	5.07%

Tabla 6.4: Porcentajes de cambio respecto a OmpSs en Hotspot + Pegaso

6.3.3 Sobel

Como ya se mencionó en la sección de implementación, se implementaron dos versiones difentes de Sobel, "Sin overlap", donde las transferencias de memorias se ejecutan en el stream por defecto de la GPU, y "Con overlap", donde cada transferencia dispone de un stream propio. Cada una de estas versiones produce diferentes resultados en función de la máquina donde se ejecute, por lo que aquí se incluyen los tiempos para ambas versiones, al no ser una claramente superior a la otra.

Todos los tiempos corresponden al mismo caso de prueba, en el que se procesan 100 fotogramas, en cada una de las versiones del programa.

Sobel + Manticore

	Ref Sync	Ref Async	Ctrl Sync	Ctrl Async	OmpSs-2	OmpSs-2 (Overlap)
File to File	0.2357	0.2087	0.2457	0.2139	0.2127	0.2836
Mem To File	0.2314	0.2024	0.2298	0.1993	0.2155	0.2917
File To Mem	0.173	13.66%	0.1784	0.1364	0.1304	0.1292
Mem To Mem	0.1661	0.1296	0.1672	0.1271	0.1484	0.1344

Tabla 6.5: Tiempos de ejecución de Sobel en Manticore

En Manticore vemos, por un lado, que la versión sin overlap mejora a las versiones síncronas en todos los casos, mientras que la versión con overlap empeora mucho los tiempos cuando se realizan escrituras en fichero, lo que lleva a tiempos peores que las versiones síncronas.

En cuanto a las versiones asíncronas los tiempos de OmpSs sin overlap son más lentos, a excepción del caso de Memoria a Memoria, que mejora los tiempos de Controllers, también hay una pequeña mejora de menos de un 1% en el caso de Fichero a Fichero. La magnitud de las diferencias varía en gran medida entre versiones de la aplicación. Los tiempos con overlap, aunque empeoran en los casos con escritura en fichero, mejoran el tiempo anterior si se escribe en memoria. En especial, hay una gran mejora en la versión de Memoria a Memoria, que pasa de ser un 12-14% más lenta que las otras versiones asíncronas a alrededor de un 3-5%.

Porcentaje de cambio en relación a OmpSs (Sobel + Manticore)

Sin Overlap	**Ref Sync**	**Ref Async**	**Ctrl Sync**	**Ctrl Async**
File to File	10.82%	-1.87%	15.52%	0.57%
Mem To File	7.40%	-6.06%	6.66%	-7.50%
File To Mem	32.71%	4.79%	36.85%	4.63%
Mem To Mem	11.94%	-12.66%	12.68%	-14.34%

Con Overlap	**Ref Sync**	**Ref Async**	**Ctrl Sync**	**Ctrl Async**
File to File	-16.89%	-26.41%	-13.37%	-24.58%
Mem To File	-20.68%	-30.62%	-21.23%	-31.68%
File To Mem	33.91%	5.73%	38.09%	5.58%
Mem To Mem	23.59%	-3.57%	24.40%	-5.43%

Tabla 6.6: Porcentajes de cambio respecto a OmpSs en Sobel + Manticore

Sobel + Pegaso

	Ref Sync	**Ref Async**	**Ctrl Sync**	**Ctrl Async**	**OmpSs-2**	**OmpSs-2 (Overlap)**
File to File	0.3456	0.2539	0.3629	0.2457	0.2918	0.2729
Mem To File	0.3265	0.2502	0.3416	0.2362	0.2842	0.2583
File To Mem	0.2635	0.1638	0.2669	0.1665	0.1945	0.1771
Mem To Mem	0.2513	0.1552	0.2569	0.1532	0.2019	0.1661

Tabla 6.7: Tiempos de ejecución de Sobel en Pegaso

Porcentaje de cambio en relación a OmpSs (Sobel + Pegaso)

Sin Overlap	**Ref Sync**	**Ref Async**	**Ctrl Sync**	**Ctrl Async**
File to File	18.43%	-12.99%	24.36%	-15.80%
Mem To File	14.89%	-11.96%	20.21%	-16.88%
File To Mem	35.45%	-15.80%	37.20%	-14.41%
Mem To Mem	24.44%	-23.15%	27.21%	-24.14%

Con Overlap	**Ref Sync**	**Ref Async**	**Ctrl Sync**	**Ctrl Async**
File to File	26.65%	-6.95%	32.99%	-9.96%
Mem To File	26.38%	-3.15%	32.23%	-8.57%
File To Mem	48.76%	-7.53%	50.68%	-6.00%
Mem To Mem	51.32%	-6.55%	54.69%	-7.75%

Tabla 6.8: Porcentajes de cambio respecto a OmpSs en Sobel + Pegaso

En Pegaso vemos que, a pesar de que se mejoran los tiempos de las versiones síncronas en la versión sin overlap, la diferencia respecto a las asíncronas es mucho mayor que en Manticore. Este comportamiento es muy diferente de lo observado para las otras dos aplicaciones, donde era precisamente en Pegaso donde se obtenían ejecuciones más rápidas.

Por otro lado, en esta máquina la versión con overlap mejora los tiempos en todos los casos. A pesar de seguir siendo más lenta que las versiones de referencia y de Controllers, en este caso se reducen considerablemente las diferencias.

6.4 Conclusiones

Las gráficas de Matrixpow y Hotspot muestran el efecto del planificador de tareas de OmpSs en las ejecuciones más rápidas. El overhead introducido anula cualquier mejora que se pueda obtener por el solapamiento de las transferencias de datos con el cómputo, y es especialmente remarcable el contraste con el overhead de Controllers, mucho menor en comparación.

En el caso de las ejecuciones con mayor carga computacional, a priori más interesantes en aplicaciones reales, vemos que las versiones de OmpSs se comportan, como era de esperar, de forma similar a las versiones con transferencias de datos asíncronas, aunque con diferencias destacables entre las dos máquinas que se han utilizado. En líneas generales, los tiempos de OmpSs en Matrixpow y Hotspot son similares a los nativos y de Controllers asíncronos en Manticore, con una ligera ventaja para los últimos, mientras que OmpSs parece tener un mejor rendimiento que estas versiones en Pegaso.

En cuanto a Sobel, el comportamiento en Manticore es similar a lo observado en las otras dos aplicaciones, obviando los tiempos de las versiones con escritura en fichero en el caso con overlap. Sin embargo, los resultados obtenidos en Pegaso no solo no mejoran a las versiones asíncronas sino que, de media, la diferencia con estas es mayor que en Manticore.

Para las pruebas de Sobel sólo se ha utilizado un caso de prueba debido a la gran cantidad de versiones del programa. Viendo cómo los resultados de OmpSs obtenidos en Manticore y Hotspot mejoran, por lo general, a medida que aumenta la carga computacional, es posible que si se realizaran pruebas con casos de prueba más grandes empezaramos a ver diferencias menores e incluso tiempos mejores por parte de OmpSs.

Es dificil estimar el efecto del scheduler en los resultados, pero a falta de otras diferencias reseñables entre el funcionamiento de las versiones de OmpSs y Controllers parece seguro asumir que esto es, al menos en parte, una de las causas cuando OmpSs produce tiempos mayores a las otras versiones.

Durante la realización de las pruebas se observó que los tiempos de ejecución en Manticore aumentaban con el número de repeticiones realizadas, en ocasiones llegando a ser el tiempo para la 30ª ejecución de un caso de prueba dado hasta un 50% mayor al de la primera.

Este comportamiento puede ocurrir a raíz de un ajuste automático de frecuencias de la GPU y la CPU si estas se calientan demasiado, como se comprobó más adelante. Para la realización de las pruebas en Manticore, se introdujeron tiempos de espera entre cada ejecución de todos los casos de prueba. En Matrixpow se observó que los resultados empezaban a ser uniformes a partir de dos minutos de espera, mientras que con Hotspot fueron necesarias esperas de cinco minutos.

Capítulo 7

Conclusiones

El objetivo de este proyecto es realizar una comparativa entre dos modelos que a priori ofrecen funcionalidades similares, para verificar cuál de ellos es más eficiente. Los cambios en la nueva versión del modelo OmpSs, sin embargo, son significativos, lo que nos deja con una comparativa algo más desajustada.

El mayor inconveniente encontrado en OmpSs-2 es la falta de las funcionalidades que ofrece OmpSs-1 en cuanto al uso de aceleradores. Por esta razón, uno de los principales motivos para realizar esta comparativa, comprobar la eficiencia de las transferencias de memoria automáticas si se usan técnicas de análisis de grafos en contraste con la aproximación de Controllers, pierde algo de peso.

En lugar de transferencias automáticas, en OmpSs-2 el programador tiene que escribir manualmente el código CUDA para esos movimientos de datos. Esto es una clara desventaja respecto a Controllers, ya que limita la portabilidad al introducirse código especifico para GPUs de Nvidia, y obliga al programador a planificar cuándo hacer estas transferencias.

Un problema relacionado es el uso de operaciones asíncronas. OmpSs-2 funciona correctamente con las llamadas a los kernels, a pesar de ser asíncronas, el planificador espera a que el kernel termine para liberar sus dependencias. Esto no es así con llamadas asíncronas a transferencias de memoria, en este caso, el planificador libera las dependencias justo después de hacer la llamada, ya que el hilo de Host vuelve inmediatamente de la llamada. Esto obliga al programador a incluir sincronizaciones de CUDA sobre los streams correspondientes.

Estos inconvenientes se deben mayormente a que OmpSs-2 está centrado en programas que se ejecutan íntegramente en CPU, y es posible que el soporte para aceleradores mejore en versiones futuras del modelo. Actualmente la solución usada por el BSC en sus ejemplos es la memoria unificada de CUDA, que mueve los datos de forma transparente para el programador. Sin embargo esta solución no se puede aplicar a programas como los que usa el grupo Trasgo, que modifican simultáneamente la misma matriz en Host y en GPU, y es, por lo general, menos eficiente que las transferencias manuales.

OmpSs-2 puede facilitar significativamente el trabajo del programador. Una de las situaciones más evidentes se presenta, precisamente, en el uso de transferencias de memoria asíncronas. Gracias al sistema de dependencias de OmpSs, para introducir una transferencia asíncrona el programador sólo tiene que incluirla dentro de una tarea, y garantizar que la transferencia ha terminado usando una sincronización explícita del stream.

Aunque esto pueda parecer complicado en comparación con un sistema que realiza estas transferencias de forma automática, lo cierto es que el esfuerzo se reduce respecto a hacer lo mismo de forma nativa. En un programa que utilizara exclusivamente CUDA para garantizar que los datos están disponibles en la memoria correspondiente antes de comenzar otras operaciones, es necesario incluir todas las esperas necesarias a través de eventos y sincronizaciones de streams. Como ejemplo, la versión asíncrona nativa de CUDA del programa matrixpow utiliza 8 sincronizaciones explícitas de CUDA en el bucle donde se realizan los cálculos. La versión de OmpSs sólo utiliza 2, las otras 6 se realizan automáticamente por el planificador de tareas.

Otra ventaja de OmpSs es que gestiona de forma automática los streams de GPU donde se envían

los kernels. A pesar de que esto sería más beneficioso si el programador pudiera desentenderse también de los streams usados para las transferencias asíncronas, es sin duda un punto a favor del modelo.

Los resultados experimentales obtenidos con las versiones finales de las aplicaciones en OmpSs-2 muestran que las diferencias entre tiempos de ejecución varían mucho en función de la aplicación y la máquina que se esté usando. Es complicado dar una única razón para estas observaciones, ya que en cada aplicación nos encontramos unas circunstancias diferentes.

En el caso de las diferencias observadas en las ejecuciones más cortas el factor principal es sin duda el overhead introducido por el planificador de tareas de OmpSs. En cuanto a las ejecuciones con mayor carga computacional, es probable que este overhead sea menos relavante en comparación con los tiempos de cómputo.

En las situaciones donde vemos diferencias inferiores a un 1% entre OmpSs, Controllers, y la referencia, podemos decir que no hay una diferencia en la práctica. Por otro lado también podemos observar casos concretos, especialmente en Pegaso, donde la diferencia es mucho mayor en ciertos casos de prueba. Para determinar si esto se debe a situaciones especiales que se den en esos casos de prueba sería necesario realizar una experimentación más extensa.

Tampoco podemos ignorar, por otro lado, los resultados obtenidos en la comparativa entre Mercuriun y Nvcc, que muestra diferencias en los tiempos de ejecución para las que no se ha encontrado una explicación definitiva. Es posible que haya ineficiencias en el compilador que estén teniendo un efecto en los tiempos.

7.1 Objetivos cumplidos

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

- Se ha estudiado el modelo OmpSs: Se ha aprendido lo suficiente acerca de OmpSs para realizar las implementaciones de los benchmarks y proponer posibles causas para los resultados obtenidos.

- Se han desarrollado versiones de los benchmarks de Matrixpow, Hotspot y Sobel correctamente paralelizadas con OmpSs: Se ha realizado la paralelización propuesta para cada una de estas aplicaciones, comprobando en todos los casos la exactitud de los resultados.

- Se ha implementado la aplicación Factorización de Cholesky con el modelo de programación CUDA para ser ejecutada en GPUs de Nvidia: Se ha conseguido una versión con CUDA de la aplicación equivalente a la de partida.

- Se ha experimentado con los benchmarks mencionados en ambos entornos, OmpSs y Controllers: Se han realizado una serie de pruebas con las versiones de OmpSs de las aplicaciones en el cluster del grupo Trasgo.

- Se han analizado los resultados obtenidos y extraído conclusiones: Se han descrito los resultados obtenidos y propuesto posibles causas para los mismos.

- Se han encontrado posibles puntos fuertes y limitaciones del modelo OmpSs: Se han encontrado una serie de ventajas y desventajas acerca del modelo, como la reducción de la complejidad a la hora de introducir transferencias asíncronas o los errores relacionados con el gestor de tareas.

7.2 Trabajo futuro

Se pretende extender la experimentación con los compiladores nvcc y Mercurium para determinar la causa de las diferencias entre las aplicaciones compiladas con cada uno, ya que esta es una de las situaciones más desconcertantes que se han observado durante el desarrollo del proyecto.

Si estas diferencias entre Mercurium y nvcc influyen en los tiempos de ejecución de los programas paralelizados con OmpSs, sería interesante tenerlo en cuenta a la hora de decidir el modelo de programación que se va a usar en un proyecto, ya que, si se quiere usar OmpSs, este es el único compilador que podremos usar.

También es importante determinar el porqué de las diferencias de tiempos de ejecución en función del número de hilos disponibles para las versiones de OmpSs. Es posible que esto esté relacionado con las propias máquinas donde se han realizado las pruebas, pero también cabe la posibilidad de que el planificador de tareas sea más ineficiente cuando dispone de muchos hilos. En este proyecto, se ha determinado experimentalmente qué número de hilos usar con cada aplicación y máquina, pero sería muy útil disponer de un método para determinar este número de antemano. Por tanto, se continuará con la experimentación en este aspecto.

Por último, se pretende implementar la factorización de Cholesky con Controllers a partir de la versión con CUDA desarrollada en este proyecto, y realizar una serie de pruebas para obtener una comparativa con las versiones originales y de CUDA.

7.3 Valoración personal

Este proyecto está más focalizado en la investigación que al desarrollo de software convencional en el que se centran muchas asignaturas del grado. Personalmente, lo tomo como una buena oportunidad para ver cómo funciona esta área de la Informática.

Por otra parte, esta investigación es diferente de lo que esperaba encontrar inicialmente. Por supuesto mi experiencia es muy limitada pero veo que, al menos en informática, gran parte del trabajo está orientado a desarrollar productos concretos que luego puedan tener aplicación en tareas de investigación en otras áreas, aunque el desarrollo de los mismos se aleje de la ingeniería de software tradicional.

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

[2] 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

[3] Cuda Programing Blog. Implementation sobel operator in cuda c on yuv video file. Web, Jan 2013. on http://cuda-programming.blogspot.com/2013/ 01/implementation-sobel-operator-in-cuda-c.html.

[4] FMA. CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/floating-point/index.html. Último acceso Julio de 2020.

[5] A. Moreton-Fernandez, H. Ortega-Arranz, and A. Gonzalez-Escribano. Controllers: An abstraction to ease the use of hardware accelerators. The International Journal of High Performance Computing Applications (IJHPCA), 32(6):838–853, 2018.

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

[7] Grupo Trasgo. Página principal. https://trasgo.infor.uva.es/. Último acceso Julio de 2020.

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

[9] Asynchronous data transfers. Nvidia developer. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/. Último acceso Julio de 2020.

[10] OmpSs. BSC Programming Models. https://pm.bsc.es/ompss. Último acceso Julio de 2020.

[11] Golub, Gene H.; Van Loan, Charles F. (1996). Matrix Computations (3rd ed.). Baltimore: Johns Hopkins. ISBN 978-0-8018-5414-9

[12] OpenMP. OpenMP main page. https://www.openmp.org/. Último acceso Julio de 2020.

[13] Centro de Supercomputación de Barcelona. Main page. https://www.bsc.es/. Último acceso Julio de 2020.

[14] OpenACC. OpenACC main page. https://www.openacc.org/. Último acceso Julio de 2020.

[15] Ompss-2 user guide. BSC Programming Models. https://pm.bsc.es/ftp/ompss-2/doc/user-guide/. Último acceso Julio de 2020.

[16] Nvprof. Nvprof user guide. https://docs.nvidia.com/cuda/profiler-users-guide/index.html. Último acceso Julio de 2020.

[17] Nvidia Visual Profiler. Nvidia developer. https://developer.nvidia.com/nvidia-visual-profiler. Último acceso Julio de 2020.

[18] Mercurium. BSC Programming Models. https://pm.bsc.es/mcxx. Último acceso Julio de 2020.

[19] Nanos. BSC Programming Models. https://pm.bsc.es/nanox. Último acceso Julio de 2020.

[20] Nanos6. Github. https://github.com/bsc-pm/nanos6. Último acceso Julio de 2020.

[21] Extrae. BSC Programming Models. https://tools.bsc.es/extrae. Último acceso Julio de 2020.

[22] Paraver. BSC Programming Models. https://tools.bsc.es/paraver. Último acceso Julio de 2020.

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

[24] Sobel, Irwin. (2014). An Isotropic 3x3 Image Gradient Operator. Presentation at Stanford A.I. Project 1968.

[25] Cholesky. BSC Programming Models. https://pm.bsc.es/gitlab/ompss-2/examples/cholesky. Último acceso Julio de 2020.

Apéndices

Apéndice A

Contenidos del Fichero ZIP

El fichero ZIP entregado contiene lo siguiente:

- Código: las versiones finales de los tres benchmarks Matrixpow, Hotspot y Sobel en OmpSs, así como las tres versiones de la factorización de Cholesky.
- Datos: los resultados de la experimentación para las aplicaciones de OmpSs.
- Enlace a la página web del grupo de investigación Trasgo, para más información o contacto.