Universidad de Valladolid

E. T. S. Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática

Desarrollo de un modelo de programación para simplificar el uso de aceleradores hardware

Autor:

Alejandro Alonso Mayo

Tutor:

Dr. Héctor Ortega Arranz
Dr. Arturo González Escribano

Agradecimientos

Quiero agradecer mi familia, quienes me han apoyado y ayudando a centrarme en el desarrollo de este trabajo. También a mis amigos quienes me apoyaban, me animaban y me escuchaban cuando necesitaba hablar con alguien. A Héctor Ortega Arranz y a Arturo González Escribano, mis tutores, quiero agradecerles por responder pacientemente dudas, aconsejarme y guiarme durante el transcurso del proyecto. Al resto del grupo Trasgo de la Uva a quienes no les faltó tiempo para ayudarme cuando tenía algún problema.

Resumen

Hoy en día el uso de aceleradores hardware, como las GPUs o las XeonPhi entre otros, está cada vez más extendido dentro del contexto de la computación de alto rendimiento. Desarrollar aplicaciones que usen estos aceleradores puede ser una tarea compleja, sobretodo si dicha aplicación conlleva una gestión no trivial de transferencias de memoria o una compleja configuración del dispositivo.

En este proyecto se pretende desarrollar una biblioteca para el desarrollo de aplicaciones que usen aceleradores hardware. Esta biblioteca pretende liberar al programador de tediosas tareas como la gestión de transferencias, configuración del acelerador y la sincronización, entre otros, a la vez que se evitan posibles fallos de una incorrecta gestión manual.

Abstract

Today the use of hardware accelerators such as GPUs or XeonPhi among others, is increasingly widespread within the context of high performance computing. Developing applications using these accelerators can be very complex, especially if the application needs a nontrivial memory transfer management or a complex configuration of the device.

This project aims to create a software development library to use hardware accelerators. This library helps the programmer with tedious tasks of programming these accelerators, such as the transferences management, and the device configuration and synchronization, among others. It also prevents possible mistakes due to incorrect manual handling.

Índice general

Índice de figuras

Índice de cuadros

Parte I

Introducción y Planificación

Capítulo 1

Introducción

En este capítulo se pretende describir la motivación para la realización de un nuevo modelo que nos simplifique el uso de los aceleradores hardware que es el propósito de este proyecto. Para ello se explicará qué son los aceleradores hardware junto con la problemática a la hora de programar aplicaciones que los utilicen.

Posteriormente se realizará una descripción de los objetivos concretos del proyecto, así como la metodología utilizada para su realización y la estructura del presente documento.

1.1. Dispositivos Aceleradores Hardware

Los aceleradores son dispositivos hardware que se añaden a un sistema informático y que permiten mejorar su rendimiento ejecutando tareas concretas de un programa que se diseñan específicamente para las arquitecturas de dichos dispositivos.

1.1.1. Introducción a los aceleradores

Las plataformas de computación de altas prestaciones incluyen cada vez más aceleradores hardware, como las GPUs, o la XeonPhi entre otros. Esta tendencia es observable desde en máquinas personales hasta en grandes plataformas de supercomputación, como se puede ver por ejemplo en las primeras posiciones de la lista de los 500 mayores supercomputadores del momento [22].

Ahora, con el uso de los aceleradores han aparecido nuevos problemas y soluciones a la hora de programar aplicaciones que aprovechen el potencial de estos dispositivos como se hablará a continuación.

1.1.2. Programación de aceleradores

La programación para plataformas que incluyen aceleradores se basa en dos posibles aproximaciones. La primera es utilizar un modelo único de programación que haga transparente en cierta medida las diferencias conceptuales entre los modelos de programación utilizados en los aceleradores y en los los "hosts" (máquinas convencionales de cómputo donde se alojan los aceleradores). En esta línea se encuentran OpenACC [3] y la propuesta de OpenMP 4.0 [4].

Sin embargo, en estos modelos las aplicaciones que no son altamente regulares y poco sincronizadas no son fáciles de expresar. Además, el código generado no contiene optimizaciones basadas en el conocimiento que tiene el programador de las características del código original que se ejecutará en los aceleradores, ni optimizaciones de parámetros que deban decidirse en tiempo de ejecución.

La segunda es construir programas mezclando modelos de programación diferentes para la computación en aceleradores y en hosts. Ejemplos de esta aproximación incluyen el uso del modelo de paso de mensajes, como por ejemplo MPI [11], con modelos de programación orientados a GPUs de fabricantes concretos, por ejemplo CUDA [2, 8]. Esta aproximación necesita un mayor conocimiento por parte del programador de los modelos, lenguajes y técnicas de programación de los diferentes aceleradores, y la tediosa labor de organizar la gestión de memoria para hacer accesibles los datos a los diferentes aceleradores en los momentos apropiados. A cambio, el programador tiene todo el control sobre los recursos de los dispositivos, pudiendo optimizar su programa para conseguir un buen rendimiento en una máquina concreta.

Además de las aproximaciones contadas anteriormente también existen múltiples propuestas intermedias. OpenCL [19] es un ejemplo de modelo en el que la mayor parte de las tareas de sincronización y manejo de estructuras de datos en un sistema heterogéneo se ocultan. Pero el programador escribe sus kernels teniendo en cuenta el modelo de multihilos agrupados, y las restricciones de sincronización propios de ciertos tipos de aceleradores. De nuevo, la genericidad implica menos capacidad de tomar decisiones automáticas sobre optimizaciones propias de cada plataforma, para obtener el máximo rendimiento. Muchas bibliotecas de funciones específicas para ciertos campos, o ciertas plataformas aceleradoras, incluyen pequeñas abstracciones para facilitar la gestión de memoria entre el acelerador y el host, pero sin capacidad de contemplar optimizaciones guiadas (e.g. MCUDA [20], o hiCUDA [7]).

1.1.3. Dificultades en la programación de aceleradores

Los aceleradores hardware tienen una serie de dificultades que complican el desarrollo de aplicaciones portables que usen dispositivos. Estas dificultades hacen que el tiempo de desarrollo se alargue, en ocasiones significativamente, y obliga a que los programadores deban tener un alto conocimiento del funcionamiento de estos aceleradores. A su vez, el grado de dificultad añadido repercute en un potencial aumento en la cantidad de errores por parte de programador, lo cual deriva en un alargamiento innecesario en el tiempo de desarrollo.

Las dificultades comunes que normalmente nos podemos encontrar durante la programación con aceleradores son las siguientes:

- *Escoger una correcta configuración del dispositivo* [9, 23]: la configuración indica al acelerador hardware parametros y condiciones en las que tiene que trabajar el dispositivo acelerador para un trabajo determinado. Una buena o mala configuración, adecuada al trabajo a realizar, tiene un importante impacto en la eficiencia del dispositivo y en el tiempo de ejecución.

- *Gestión de transferencias* [9]: La mayor parte de los dispositivos no pueden acceder directamente a datos que no estén en su espacio de memoria, o si es posible el acceso es muy ineficiente. Esto obliga a que cuando un acelerador necesita un conjunto de datos, estos tengan que ser transferidos previamente a su memoria local, a donde sí puede acceder. Lo mismo ocurre con los resultados de una operación, los cuales se

escriben en la memoria local y después se transfieren donde sean necesarios (el host u otro dispositivo).

- *Adaptación de los programas* [9]: algunos aceleradores hardware, especialmente las GPUs, usan un modelo de programación diferente al tradicional de CPU. Por este motivo, cuando se desea utilizar un programa CPU (secuencial o paralelo) en una GPU es necesario un proceso de adaptación del programa. Dicho proceso de adaptación tiene que tener en cuenta las características de la plataforma a la que se está adaptando, para poder realizar optimizaciones dependientes de la plataforma los cuales pueden tener un gran impacto en el rendimiento.

1.2. Objetivos del proyecto

Como se ha introducido, la programación de aceleradores hardware tiene una serie de dificultades específicas. Estas dificultades obligan a gastar tiempo focalizándose en detalles que están fuera del objetivo de la aplicación que pretenden desarrollar, alargando consecuentemente el tiempo de desarrollo de dichas aplicaciones. A su vez, dichas dificultades, pueden llevar a que el programador, en caso de que no este muy versado en el uso de estos aceleradores o que nunca los haya utilizado, cometa errores, especialmente de configuración y gestión de transferencias.

Este proyecto tiene como principal objetivo buscar una solución a dichos problemas. Para ello se pretende desarrollar un modelo, y un prototipo que verifique este modelo, que cumpla los siguientes objetivos:

- Configuración automática del dispositivo: Si el programador lo desea, por desconocimiento o para reducir el tiempo de desarrollo, debe poder delegar a nuestro modelo la configuración de parámetros y condiciones de ejecución del dispositivo de forma automática.
- Transferencias transparentes al programador: El modelo debe administrar las transferencias sin que el programador tenga que conocer los mecanismos necesarios para que los datos contenidos en las estructuras de datos a transferir lleguen al dispositivo.
- Unificar distintos paradigmas de programación heterogénea: Idear una forma de programación que acerquen los paradigmas de programación entre distintos tipos de aceleradores, permitiendo a los programadores reducir la necesidad de adaptación de los algoritmos o programas.

1.3. Metodología utilizada

Este proyecto es un trabajo de investigación en el que se pretende buscar una nueva solución a un problema existente. La investigación no es un trabajo rectilíneo hacia la solución si no que requiere de una evolución constante, donde una vez se ha llegado a la última etapa es posible encontrar posibles mejoras adicionales para la nueva solución. En este tipo situaciones y otras parecidas se necesita un modelo de proceso diseñado explícitamente para adaptarse a un producto que evoluciona con el tiempo. Además la metodología tiene que ser una metodología de investigación la cual difiere ligeramente del las metodologías para el desarrollo de software convencional.

Para cumplir con los requisitos de metodología de este trabajo, éste ha sido basado en el paradigma de hacer prototipos [16] y en el método de investigación en ingeniería del software [1]. El paradigma de hacer prototipos consiste en una metodología evolutiva en el cual durante una serie de iteraciones donde para cada iteración se realizan todos los pasos de: obtención de requisitos, planificar la iteración, análisis y diseño del prototipo, construcción del prototipo y, por último, entrega y evaluación. El método de investigación en ingeniería del software consiste en una metodología donde la investigación se divide en 4 partes diferenciadas: observar las soluciones existentes, proponer mejores soluciones, construir o desarrollar esa nueva solución y analizar sus resultados. Obteniendo así una metodología de investigación basada en prototipos la cual es evolutiva.

Por tanto, la metodología utilizada en este trabajo consiste en realizar una serie de iteraciones donde en cada iteración se realizarán los siguientes pasos: observar las soluciones existentes, planificar la iteración, proponer mejores soluciones, análisis y diseño del prototipo, construcción del prototipo y analizar sus resultados.

1. Observar las soluciones existentes: Se trata de una fase exploratoria, donde la documentación relacionada será analizada a fondo con el fin de detectar no sólo las limitaciones que serán abordarse durante el proceso de investigación, sino también posibles mejoras y/o nuevas soluciones aún no contempladas.

2. Planificar la iteración: En esta fase se fijarán plazos para las distintas fases de la iteración, asi como los hitos importantes. Esta fase es de suma importancia puesto que asegura que el proyecto entregue resultados en las fechas establecidas.

3. Proponer soluciones mejores: Esta fase está dedicada al análisis y diseño de encontrar una mejor solución tratando de superar los límites u obtener ventaja de las posibles mejoras detectadas previamente.

4. Análisis y diseño del prototipo: En esta fase se realizará el análisis y diseño del prototipo a partir de la propuesta de solución obtenida.

5. Construcción del prototipo: Durante esta fase se realiza la implementación del prototipo junto con las pruebas que validen la buena construcción del mismo.

6. Analizar la nueva solución: Los prototipos implementados de las soluciones se evalúan empíricamente, con el fin de corroborar si resuelven los problemas descubiertos en la primera fase.

1.4. Estructura del documento

Esta memoria se encuentra dividida en 5 grandes partes: introducción y planificación, conocimientos previos y propuestas, análisis y diseño, implementación y pruebas, y experimentación y conclusiones.

La primera y presente parte corresponde a la introducción y planificación. Esta parte continuará explicando, en el Capítulo 2, la planificación realizada para el desarrollo del proyecto y posteriormente, en el Capítulo 3, el seguimiento de dicha planificación.

A lo largo de la segunda parte se introducirán diferentes tecnologías necesarias para comprender este proyecto. En el Capítulo 4 se hace una introducción al funcionamiento de

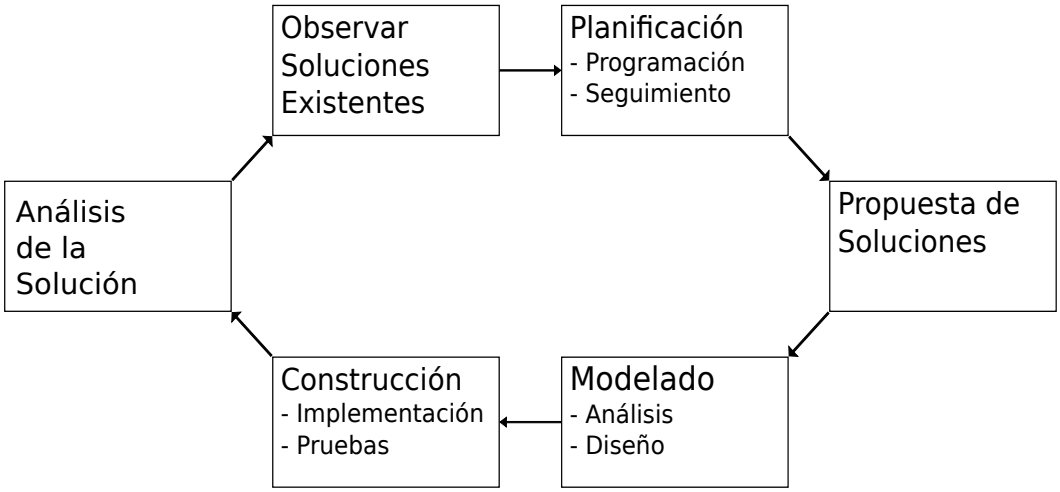

Figura 1.1: Gráfico explicativo de la metodología utilizada.

los dispositivos CUDA y a su modelo de programación permitiendo así entender mejor la motivación de este proyecto. En el Capítulo 5 se inroduce la biblioteca Hitmap usada en la implementación del prototipo del proyecto.

En la tercera parte se realizará un análisis y diseño del proyecto. Primero, en el Capítulo 6, se desarrollará el modelo teórico, el cual posteriormente se desarrollará. En el Capítulo 7 se realizará un pequeño análisis del modelo desarrollado, del cual se obtendrán: los requisitos, los casos de uso y el modelo de objetos; para el desarrollo de un prototipo donde realizar la experimentación. Por último, en el Capítulo 8 se realizará el modelo del diseño del prototipo donde se decidirá su estructura y su forma de actuar.

La cuarta parte se especificará como se ha realizado la implementación, en el Capítulo 9, y las pruebas que se le han realizado al prototipo, en el Capítulo 10). En la implementación se especificarán las herramientas utilizadas, posibles alternativa y porqué se escogió dicha herramienta. Además se explicará como se realizó la implementación de las partes más importantes del prototipo. En las pruebas se especificará la batería de pruebas realizadas y sus correspondientes resultados.

En la quinta y última parte se realizará una serie de experimentos, en el Capítulo 11, con los que se desea medir el rendimiento del modelo y el esfuerzo de programar utilizando el modelo elegido. A partir de la experimentación se obtendrán una serie de resultados que se utilizarán en el Capítulo 12 para sacar conclusiones sobre el modelo y el propio proyecto.

Capítulo 2

Planificación del proyecto

En este capítulo se va a describir la planificación del proyecto. La planificación de un proyecto consiste en el planteamiento, la organización y el control de los recursos con motivo de alcanzar los objetivos del mismo.

Este capítulo se divide en cinco secciones. En la primera sección se explican los artefactos más importantes que conforman el proyecto. En la segunda sección se especifica cómo se va a gestionar el tiempo del que se dispone para la realización del proyecto. En la tercera sección se especifica como se va a gestionar el coste del proyecto. En la cuarta sección se listan los riesgos y cómo se debe actuar en caso de que ocurran. Por último, en la quinta sección se especifica como se va a realizar la gestión de configuraciones, es decir, almacenamiento de los artefactos, control de cambios, etc.

2.1. Entregables del proyecto

Durante el desarrollo del proyecto se generan y utilizan una serie de artefactos, quienes son objeto de modificaciones. Al término de este desarrollo, dichos artefactos conforman la documentación del proyecto, la cual está recopilada en la presente memoria. Los artefactos que pertenecen a este proyecto son:

- Gestión del Proyecto: son los documentos relacionados con la planificación y el seguimiento del proyecto. Esta documentación consta de los siguientes artefactos:
  - Planes de iteración (S. 2.2.1): Es el conjunto de todas las actividades pertenecientes a una iteración programadas con sus dependencias.
  - Plan de riesgos (S. 2.4): Este documento incluye una lista de los riesgos conocidos y vigentes en el proyecto junto con las acciones específicas a realizar en caso de que ocurra.
  - Seguimiento del proyecto (C. 3): Este documento contiene los cambios que se producen sobre lo planificado e intenta evitar situaciones críticas como consecuencia de los mismos.
- Análisis y diseño: Son los documentos relacionados con el desarrollo del sistema. Contiene los siguientes artefactos.

    - Modelo propuesto (C. 6) Contiene el modelo teórico propuesto en este proyecto que será el que se desarrolle. Aquí se explica como funciona y sus capacidades.
    - Modelo de Análisis (C. 7) Contiene una visión inicial del comportamiento del sistema, así como la lista de requisitos que debe cumplir, los casos de uso y el modelo de objetos.
    - Modelo de Diseño (C. 8) Contiene los modelos del diseño que se va a implementar en el prototipo, detallando la estructura estática del sistema.

- Implementación: contiene toda la documentación relativa a la implementación del proyecto. Consta de los siguientes artefactos:

    - Modelo de implementación (C. 9): Contiene una descripción de los detalles de la implementación del prototipo. Especialmente las herramientas utilizadas y cómo se han resuelto los problemas encontrados.
    - Pruebas (C. 10): Contiene la especificación de cada una de las pruebas realizadas, junto con su entrada y su salida esperada. Además, contiene las tablas con los resultados de dichas pruebas.
    - Código fuente (CD-ROM, ver anexo A)

- Experimentación (C. 11): Contiene la especificación y los resultados de los experimentos realizados sobre el prototipo creado.

2.2. Gestión de tiempo

Durante el desarrollo de un proyecto con un plazo de tiempo limitado es importante tener en cuenta en qué se usa el tiempo. Una buena gestión del tiempo permite realizar el proyecto en la mayor brevedad posible, al paralelizar trabajos por ejemplo. La gestión del tiempo también permite reducir las repercusiones de un retraso en una tarea.

2.2.1. Planes de iteración

Plan iteración 1

En la primera iteración se tiene como objetivo crear una versión del prototipo con la interfaz que debiera tener una biblioteca que implementa nuestro modelo. Para lograr esto, es necesario realizar un estudio del estado del arte y de tecnologías relacionadas con este proyecto. A partir de esto, se pensará en varias propuestas de modelo de entre las cuales se seleccionará y desarrollará uno, a partir del cual se desarrollará un prototipo.

El prototipo en esta iteración será una interfaz sin ningún tipo de funcionalidad. Esta interfaz sin funcionalidad constará de una serie de funciones que no realizan ninguna acción, que al aplicarla sobre un problema, nos permitirá medir el esfuerzo de desarrollo y comprobar así si nuestro modelo simplifica la programación.

Para las pruebas se utilizará un problema frecuente en computación de alto rendimiento programado tanto usando la nueva interfaz como sin usarla. Esto nos permitirá comparar el esfuerzo de desarrollo, es decir, el coste de programarlo, aunque el prototipo no sea funcional.

La lista de actividades de la primera iteración se puede observar en el Cuadro 2.1, asi como los diagramas de Gantt correspondientes en las Figuras 2.1 y 2.2.

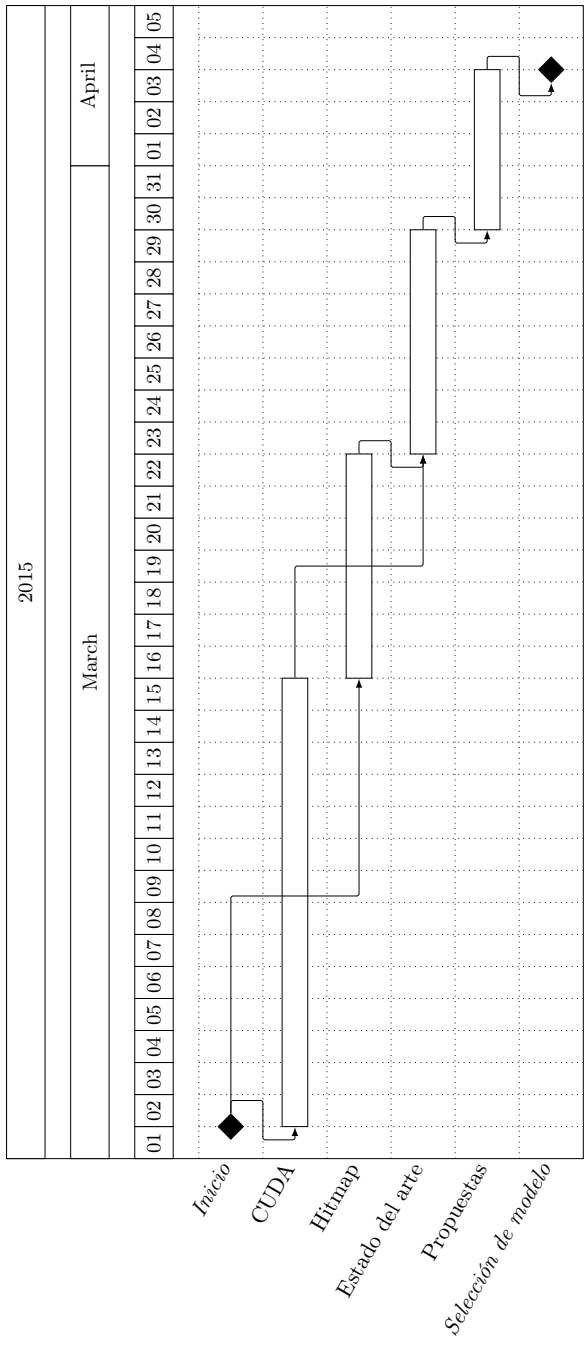

Figura 2.1: Diagrama de Gantt Iteración 1

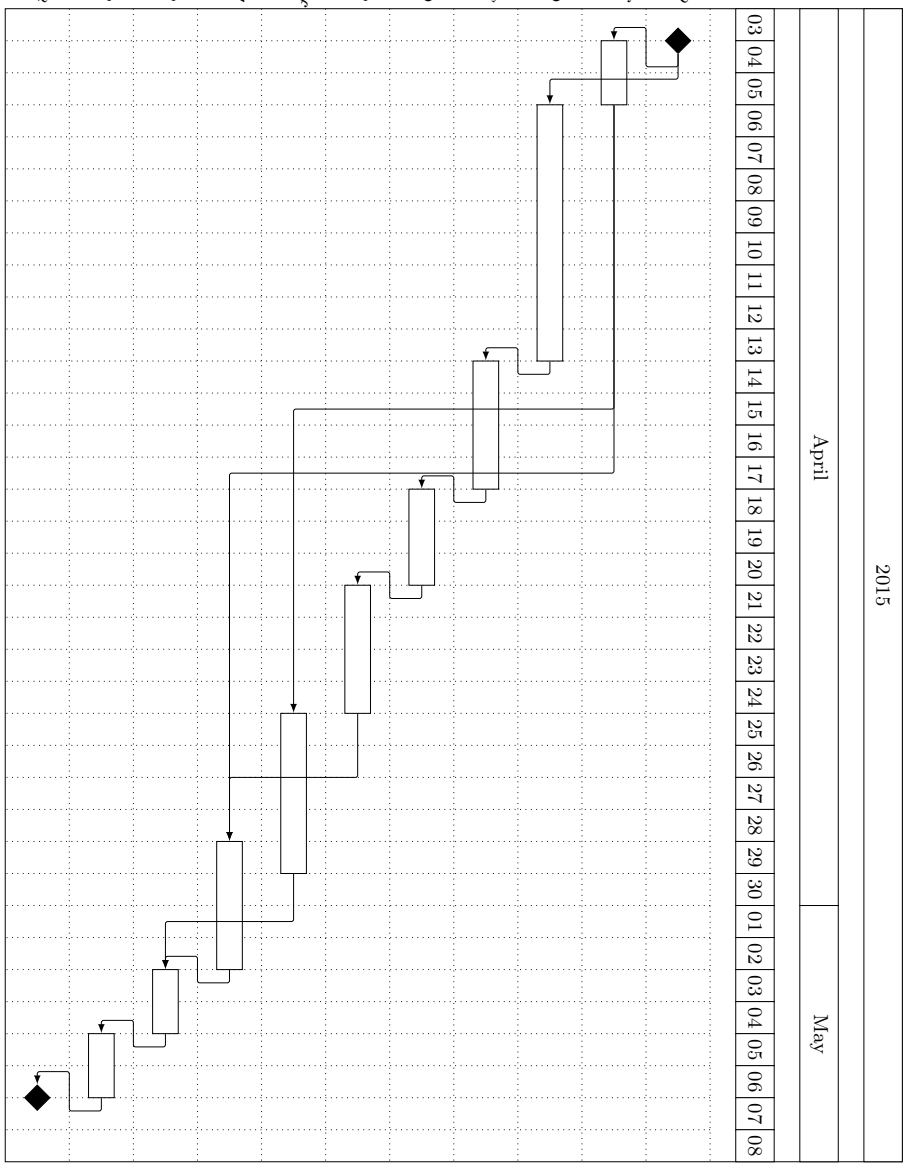

Figura 2.2: Diagrama de Gantt Iteración 1 (cont)

Nº	Nombre de la actividad	Fecha inicio	Fecha fin	Actividades Precedentes
1	Inicio	2/03/15	2/03/15	
2	Aprendizaje de CUDA	2/03/15	15/03/15	1
3	Aprendizaje de Hitmap	16/03/15	22/03/15	1
4	Exploración del estado del arte	23/03/15	29/03/15	2,3
5	Propuesta de posibles modelos	30/03/15	3/04/15	4
6	Selección de modelo	4/04/15	4/04/15	5
7	Búsqueda de problemas de referencia	4/04/15	5/04/15	6
8	Desarrollo del modelo	6/04/15	13/04/15	6
9	Análisis del prototipo	14/04/15	17/04/15	8
10	Diseño del prototipo	18/04/15	20/04/15	9
11	Construcción del prototipo	21/04/15	24/04/15	10
12	Construcción del experimento Jacobi: GPU vRef	25/04/15	29/04/15	7
13	Construcción del experimento Jacobi: GPU v1	30/04/15	2/05/15	7,11
14	Experimentación	3/05/15	4/05/15	12,13
15	Planificación siguiente iteración	5/05/15	6/05/15	14
16	Fin	7/05/15	7/05/15	15

Cuadro 2.1: Actividades de la primera iteración

Plan iteración 2

En la iteración anterior se demostró que el uso de los comunicadores realmente reducía el esfuerzo en el desarrollo. Para la segunda iteración se revisará el modelo añadiendo las modificaciones que se consideren oportunas a partir de lo ya observado en la iteración anterior.

Junto con las modificaciones en el modelo, se deberá realizar un prototipo funcional que implemente la mayor parte de las funcionalidades que se plantean. Este prototipo se utilizará para realizar mediciones de rendimiento que nos permitirán determinar el overhead en tiempo de ejecución que se produce con el uso de este modelo. Para medirlo en un caso práctico, se utilizará el mismo problema de la iteración anterior pero adaptado a la nueva versión del prototipo. Así, comparando los resultados con los de la versión de referencia podremos ver la diferencia en el rendimiento. En esta iteración se volverá a medir el esfuerzo de desarrollo con el fin de comprobar que los cambios realizados en el modelo no aumentan el esfuerzo de desarrollo obtenido en la primera iteración. Los resultados de la experimentación de esta iteración servirán para la preparación de un articulo, incluido en la planificación, para las Jornadas de Paralelismo.

La lista de actividades de la segunda iteración se puede observar en el Cuadro 2.2, asi como el diagrama de Gantt la Figura 2.3.

Plan iteración 3

Para la tercera iteración se modificará el modelo añadiendo la posibilidad de poder tratar un conjunto de núcleos de la CPU como si fuera un acelerador hardware externo. Posteriormente se deberá modificar el prototipo para que se adapte de nuevo al modelo, junto con

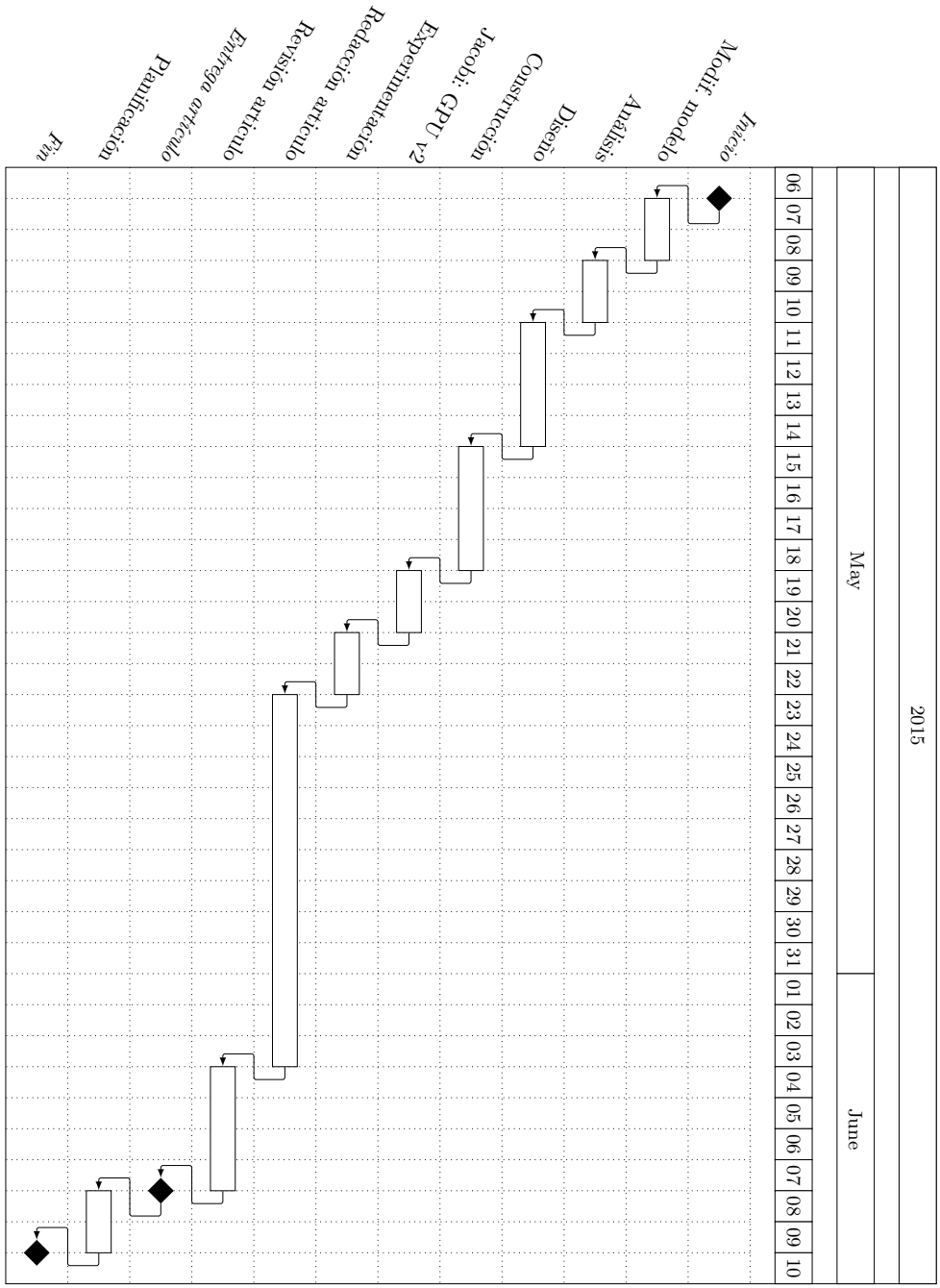

Figura 2.3: Diagrama de Gantt Iteración 2

Nº	Nombre de la actividad	Fecha inicio	Fecha fin	Actividades Precedentes
1	Inicio	7/05/15	7/05/15	
2	Modificación del modelo	7/05/15	8/05/15	1
3	Análisis del prototipo	9/05/15	10/05/15	2
4	Diseño del prototipo	11/05/15	14/05/15	3
5	Construcción del prototipo	15/05/15	18/05/15	4
6	Construcción del experimento Jacobi: GPU v2	19/05/15	20/05/15	5
7	Experimentación	21/05/15	22/05/15	6
8	Redacción del artículo para las Jornadas de Paralelismo	23/05/15	3/05/15	7
9	Revisión del artículo	4/06/15	7/06/15	8
10	Entrega del artículo	8/06/15	8/06/15	9
11	Planificación de la siguiente iteración	8/06/15	9/06/15	10
12	Fin	10/06/15	10/06/15	11

Cuadro 2.2: Actividades de la segunda iteración

los experimentos ya creados para que se adapten al mismo. Así mismo, se deberán obtener versiones de referencia de CPU para un conjunto más amplio de aplicaciones incluyendo Jacobi, la suma de matrices y la multiplicación de matrices, al igual que versiones de referencia de GPU para la suma de matrices y la multiplicación de matrices. Además, también se deberán construir versiones de los experimentos usando nuestro prototipo. En esta iteración se medirá el esfuerzo de desarrollo, el overhead y el coste de migrar una versión de GPU a una de CPU y viceversa. Los resultados de la experimentación de esta iteración servirán para la preparación de un articulo, incluido en la planificación, para el workshop HLPGPU celebrado dentro de la conferencia HiPEAC.

La lista de actividades de la segunda iteración se puede observar en el Cuadro 2.3, así como el diagrama de Gantt la Figura 2.4.

2.3. Costes del proyecto

Las estimaciones de costes son necesarias para establecer un presupuesto para el proyecto o para asignar un precio para el software de un cliente. Existen tres parámetros involucrados en el cálculo del coste total de un proyecto de desarrollo de software.

- Los costes hardware y software, incluyendo el mantenimiento.
- Los costes de viaje y capacitación.
- Los costes de esfuerzo.

Tanto en este proyecto como en otros muchos, los costes dominantes son los costes de esfuerzo. Los ordenadores con potencia suficiente para desarrollar software son relativamente baratos. Por otro lado, el software necesario para el desarrollo de la mayoría del Proyecto ha sido gratuito, dado que todos poseen licencia libre. Aunque haya costes de viaje, son una pequeña parte comparados con los costes de esfuerzo. Además, el uso de correo electrónico,

Nº	Nombre de la actividad	Fecha inicio	Fecha fin	Actividades Precedentes
1	Inicio	1/10/15	1/10/15	
2	Modificación del modelo	1/10/15	4/10/15	1
3	Análisis del prototipo	5/10/15	6/10/15	2
4	Diseño del prototipo	7/10/15	12/10/15	3
5	Construcción del prototipo	13/10/15	17/10/15	4
6	Construcción del experimento Jacobi: CPU vRef	18/10/15	19/10/15	1
7	Construcción del experimento Jacobi: CPU v1	20/10/15	20/10/15	5
8	Construcción del experimento MatrixAdd: CPU y GPU vRef	21/10/15	22/10/15	1
9	Construcción del experimento MatrixAdd: CPU y GPU v1	23/10/15	24/10/15	5
10	Construcción del experimento MatrixMult: CPU y GPU vRef	25/10/15	26/10/15	1
11	Construcción del experimento MatrixMult: CPU y GPU v1	27/10/15	28/10/15	5
12	Experimentación	29/10/15	30/10/15	6,7,8,9,10,11
13	Redacción del artículo para HLPGPU	31/10/15	11/11/15	12
14	Revisión del articulo	12/11/15	15/11/15	13
15	Entrega del articulo	16/11/15	16/11/15	14
16	Fin	16/11/15	16/11/15	15

Cuadro 2.3: Actividades de la tercera iteración

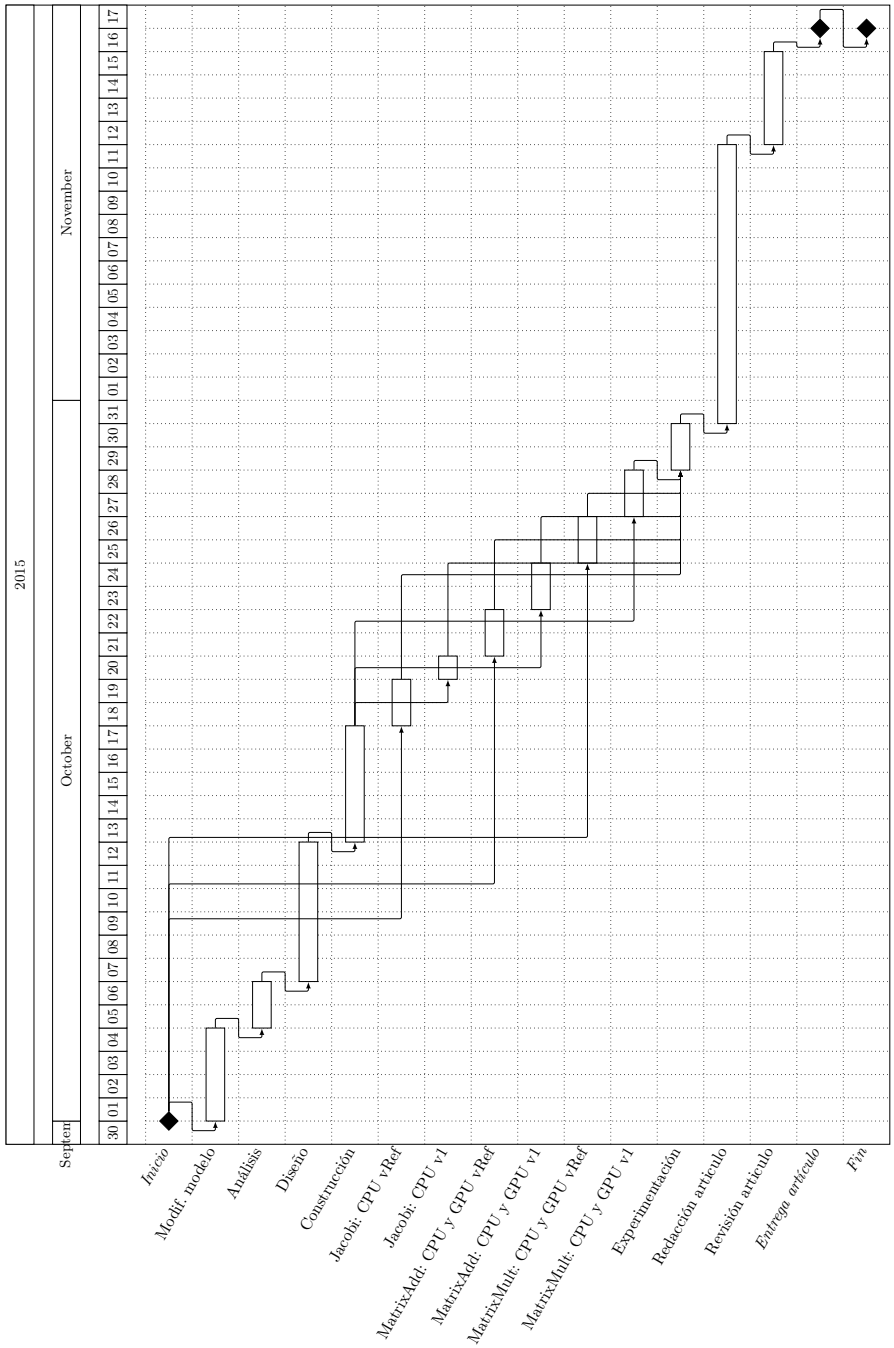

Figura 2.4: Diagrama de Gantt Iteración 3

Concepto	Cantidad	Precio (€)
Herramientas de desarrollo		
Ubuntu Linux 14.10	1	0,00
CMake v2.8.11	1	0,00
GCC v4.8.3	1	0,00
GDB v7.6.1	1	0,00
VIM v7.4	1	0,00
GanttProject v2.7.1	1	0,00
Inkscape v0.91	1	0,00
LaTeX	1	0,00
DropBox	1	0,00
GitLab	1	0,00
Subtotal		**0,00**
Máquinas		
Acer Travelmate 5740-334G32MN	1	496,00
Subtotal		**496,00**
Alquiler Máquinas		
Intel Xeon CPU E5-2620 (24 CPUs) con tarjeta gráfica Nvidia GeForce GTX Titan Black	384 ($h * cpu$)	2,00
Subtotal		**768,00**
Alquiler Servidores		
VPS	12 *meses*	10,00
Subtotal		**120,00**
Horas de trabajo		
Número de horas	1168	12.50
Subtotal		**14.600,00**
Total		**15.984,00**

Cuadro 2.4: Presupuesto estimado

teléfono y sitios web compartidos reducen el coste de los viajes y del tiempo hasta en un 50 %.

Los costes de esfuerzo no son sólo los salarios de los ingenieros que intervienen en el proyecto. Las organizaciones calculan los costes de esfuerzo en función de los costes totales, donde se tiene en cuenta el coste total para hacer funcionar la organización y dividen éste entre el número de personas productivas. Por lo tanto, los siguientes costes son parte de los costes totales:

1. Costes de proveer, aclimatar e iluminar oficinas.
2. Los costes del personal de apoyo como administrativos, secretarias, limpiadores y técnicos.
3. Los costes de redes y de comunicaciones.
4. Los costes de los recursos centralizados como las bibliotecas, los recursos recreativos, etc.
5. Los costes de seguridad social, pensiones seguros privados, etc.

Según Sommerville [18], este factor de carga normalmente es el doble del salario de un ingeniero del software, dependiendo del tamaño de la organización y sus sobrecargas asociadas. Sin embargo, como en nuestro proyecto todos los costes mencionados corren a cargo de la Universidad de Valladolid, se ha decidido no tenerlos en cuenta a la hora de elaborar un presupuesto. Por tanto, teniendo todo esto en cuenta, el coste aproximado sería de: **15.984 €**. En el Cuadro 2.4 se puede ver un desglose de los costes estimados de este proyecto. Hay que tener en cuenta que el coste de alquiler del cluster usado para experimentación se calcula como $CosteAlquiler = HorasComputacion * NumeroProcesadores * Precio$ donde se han calculado unas 8 horas de computación por iteración (segunda y tercera iteración). El precio de alquiler por hora se estima en función de costes medios habituales en centros de supercomputación regionales o del CIEMAT.

2.4. Gestión de riesgos

Durante el desarrollo de un proyecto pueden surgir contratiempos que afecten al proyecto provocando pérdidas y retrasos. Un riesgo es un posible problema que tiene potencial para producirse en el proyecto. Para evitar los riesgos se va a realizar una lista con todos los posibles casos y crear un plan de acción para cada uno en caso de que surja.

2.4.1. Perspectiva general

Los riesgos que se listan a continuación se describen mediante una serie de campos:

- **Probabilidad:** Estimación de la probabilidad de que se produzca el riesgo. Los posibles valores que puede tener este campo según su probabilidad: muy baja (<= 0,2), baja (> 0,2 y <= 0,4), media (> 0,4 y <= 0,6), alta (> 0,6 y <= 0,8) y muy alta (> 0,8).
- **Consecuencias:** Indica el nivel de magnitud e importancia que se le da a las posibles consecuencias que producirá el riesgo en caso de que se haga realidad. Los posibles valores son: despreciable (<= 0,2), marginal (> 0,2 y <= 0,4), crítico (> 0,4 y <= 0,8) y catastrófico (> 0,8).

- **Descripción:** Contienen una descripción del riesgo indicando como podría producirse.
- **Contexto:** Contiene una explicación del contexto en el que se puede desarrollar el riesgo descrito. Principalmente describe cuándo, cómo y por qué puede producirse un riesgo.
- **Análisis:** Se hace una breve descripción de las consecuencias que podrían producirse en caso de que el riesgo se haga realidad.
- **Estrategia de resolución:** Tipo de estrategia que se utilizará para solventar el problema en caso de que se produzca un riesgo. Estas estrategias podrán ser:
  - Evasión: previene la ocurrencia del riesgo reduciendo su probabilidad a cero.
  - Protección: reduce la probabilidad y/o consecuencia del riesgo antes de que ocurra.
  - Reducción: reduce la probabilidad y/o consecuencia del riesgo después de que ocurra.
  - Investigación: obtener más información para eliminar o reducir la incertidumbre
  - Reserva: utilizar la planificación reservada previamente o la holgura del presupuesto.
  - Transferencia: reorganizar las cosas para desplazar el riesgo a cualquier parte (por ejemplo, a otro grupo)

  Hay que considerar además la aceptación del riesgo que se produce cuando el coste de la evitación del riesgo puede ser más grande que el coste que puede suponer si se produce.
- **Plan de acción:** Descripción de las medidas que se tomarán en caso de que se produzca el riesgo.

2.4.2. Lista de riesgos

R-1. Cambios en la propuesta modelo

Probabilidad: Depende de la iteración:

- Primeras iteraciones: Alta (0.8)
- Últimas iteraciones: Baja (0.4)

Consecuencias: Depende de la iteración:

- Primeras iteraciones: Marginal (0.3)
- Últimas iteraciones: Crítico (0.7)

Descripción: El modelo teórico, que contiene la solución propuesta, puede ser modificado. Estas modificaciones pueden ser mejorar la descripción de alguno de los conceptos que introduce o añadir un nuevo concepto alterando drásticamente el modelo.

Contexto: Durante el desarrollo del prototipo, los casos de prueba e incluso la experimentación, se pueden detectar errores o posibles mejoras en la solución propuesta. Dichos errores o propuestas pueden llevar a una modificación del modelo teórico añadiendo o modificando alguno de los conceptos del mismo.

Análisis: Una modificación del modelo, dependiendo de la gravedad de las modificaciones, puede tener grandes consecuencias. Esto se debe a que las modificaciones en el modelo implica que de debe modificar todo aquello que dependa de él, provocando graves retrasos en el desarrollo del prototipo.

Estrategia de resolución: Investigación y Reducción

Plan de acción: En caso de que se detecte una posible mejora, se deberá estudiar las consecuencias de introducir dicho cambio en el modelo. En el estudio se tendrá en cuenta si el cambio está dentro del marco de los objetivos del proyecto, el coste de modificar el prototipo y el coste de realizar la experimentación necesaria añadida. Una vez realizado este estudio se decidirá si se aplicará dicho cambio al modelo y la iteración, la presente o la siguiente, en la que se realizará el cambio.

R-2. Cambios en los requisitos

Probabilidad: Depende de la iteración:

- Primeras iteraciones: Alta (0.9)
- Últimas iteraciones: Baja (0.4)

Consecuencias: Depende de la iteración:

- Primeras iteraciones: Despreciable (0.1)
- Últimas iteraciones: Crítico (0.7)

Descripción: Durante el desarrollo del proyecto cabe la posibilidad de que ocurran algunos cambios en los requisitos, tanto los funcionales como los no funcionales.

Contexto: En las distintas etapas del desarrollo del prototipo es posible que se detecte una nueva funcionalidad que deba ser añadida, o tipos de problemas a los que se puede aplicar el proyecto que no se han contemplado con anterioridad, incurriendo en la posible necesidad de un cambio en los requisitos. En cualquier caso puede resultar relevante añadirlos en el marco de desarrollo del prototipo puesto que pueden permitir adaptarse mejor al modelo teórico planteado.

Análisis: El impacto de los cambios de requisitos puede afectar significativamente al desarrollo del prototipo incrementando el coste y tiempo de desarrollo. Estos cambios pueden hacer que se cambie el diseño y la implementación e incluso ambos, además de obligar a volver a realizar la experimentación. Dicha avalancha de cambios, cuya gravedad depende del estado de desarrollo del prototipo, puede provocar que el proyecto no sea terminado en el plazo establecido.

Estrategia de resolución: Investigación y Reducción

Plan de acción: Una vez que se ha detectado un posible cambio en los requisitos del prototipo se analizará el impacto de realizar ese cambio concreto. Dependiendo del impacto y la fase en la que se ha detectado se actuará de la siguiente forma: en las primeras fases de la iteración, siempre se realizarán los cambios; sin embargo, en las etapas finales se realizará únicamente cuando el impacto sea trivial o su realización sea crítica. Cuando no se realiza un cambio en la iteración actual, estos cambios se realizarán en la siguiente iteración.

R-3. No completitud o ambigüedad de los requisitos

Probabilidad: Baja (0.4)

Consecuencias: Crítico (0.7)

Descripción: Los requisitos que fueron analizados que se están utilizando como base en nuestro prototipo, pueden no estar completos o no entenderse claramente.

Contexto: Durante el desarrollo del prototipo, se utilizan los requisitos para saber qué funcionalidad se debe implementar. Los requisitos se obtienen a partir de la propuesta del modelo desarrollada previamente, la cual puede no ser comprendida completamente a la hora de realizar los requisitos. A su vez, los requisitos pueden tener una mala redacción lo cual puede derivar en confusión a la hora de interpretarlos.

Análisis: Esto puede resultar en que el prototipo no cumpla completamente con los objetivos impuestos y no sirva para la experimentación que demuestre la eficacia del modelo teórico.

Estrategia de resolución: Investigación y Reducción

Plan de acción: Lo primero que se debe realizar es un análisis de la importancia en cuanto a los requisitos ambiguos, a la vez que se analiza el impacto en el cambio de éstos. Aquellos requisitos con gran importancia, independientemente del impacto que produzca, o con impacto despreciable serán modificados o añadidos en la misma iteración. Para los requisitos con menor importancia, pero con impacto apreciable, sus cambios se aplicarán en iteraciones posteriores.

R-4. Mal diseño

Probabilidad: Media (0.3)

Consecuencias: Marginal (0.4)

Descripción: El modelo de diseño realizado puede no tener el nivel de detalle suficiente, dejando muchos puntos abiertos, o mostrar incoherencias.

Contexto: Al comenzar las tareas de implementación, o durante su realización, faltan detalles en el diseño teniendo que tomar en este momento decisiones que ya deberían haber sido tomadas. Esto puede surgir debido a que el diseñador no ha tenido en cuenta completamente todos los escenarios posibles, o se le ha pasado por alto algún detalle crucial.

Análisis: Un diseño incompleto, o con incoherencias, puede afectar de forma grave al desarrollo del proyecto, pues se desplaza parte de la carga de trabajo de la fase de diseño a la de implementación alargando el mismo y produciendo grandes retrasos no planificados en éste. Esto se magnifica si la parte incompleta del diseño afecta, en mayor o menor medida, a otra parte del diseño impidiendo su implementación.

Estrategia de resolución: Reducción

Plan de acción: Lo que se debe hacer en este caso es modificar las partes del diseño que sean vitales y nos permitan realizar un prototipo funcional. Estas modificaciones no tienen que ser el mejor diseño posible pero si tienen que ser un diseño suficiente para que el prototipo sea válido. Todos aquellos errores encontrados, tanto los arreglados como los que no, se tendrán en cuenta durante la fase de diseño de la iteración siguiente.

R-5. Retraso en la finalización de las actividades

Probabilidad: Media (0.5)

Consecuencias: Crítico (0.5)

Descripción: El desarrollo de las actividades termina fuera del plazo establecido para ello.

Contexto: Durante el desarrollo de una actividad, pueden surgir una serie de contratiempos que finalmente pueden ocasionar un retraso en la actividad. Esto puede ser debido a eventos inesperados, que se produzcan riesgos o recursos insuficientes.

Análisis: El retraso en la finalización de actividades supone un problema respecto al cumplimiento de los plazos de las actividades siguientes, no sólo las que dependen de esta, si no también de las que no, puesto que el equipo de desarrollo tiene un número limitado de recursos humanos. Cuanto más adelante ocurra, peores son las consecuencias puesto que más cercana se encuentra la fecha de entrega y se dispone de menos tiempo para subsanar las consecuencias.

Estrategia de resolución: Reserva y Reducción.

Plan de acción: En caso de que se produzca el riesgo será necesario hacer un reajuste en la planificación empleando las posibles holguras para comprobar la gravedad del tiempo perdido. En caso de que se produzca un retraso grave o cualquier nivel de gravedad en fechas cercanas a la fecha de entrega, se aumentará el tiempo dedicado al proyecto (horas extra) para recuperar el tiempo perdido, con su consiguiente aumento en el coste del proyecto.

R-6. Errores en bibliotecas de terceros

Probabilidad: Media (0.4)

Consecuencias: Marginal (0.3)

Descripción: Durante la fase de construcción se detecta un error en una biblioteca que no pertenece al proyecto.

Contexto: Mientras se está implementando o en las pruebas, es posible que se detecten incoherencias en los resultados de una función de una biblioteca de terceros. Esto puede ocurrir con mucha frecuencia puesto que, entre otros, las bibliotecas que se usan son prototipos de otros equipos de investigación.

Análisis: Esto puede producir un retraso en las actividades que dependan de la funcionalidad con errores de estas bibliotecas de terceros.

Estrategia de resolución: Reducción y Transferencia.

Plan de acción: En el caso de que ocurra el riesgo, la primera acción a tomar es intentar evitar esa funcionalidad usando funciones similares de la misma biblioteca o de otras y reportar el error al equipo de desarrollo de la otra biblioteca. En caso de no ser posible evitar su uso se intentará implementar por cuenta propia dicha funcionalidad. Por último, si no ha sido posible implementar esa funcionalidad por cuenta propia tan sólo queda ponernos en contacto con el equipo de desarrollo de la biblioteca.

R-7. Pérdida de datos

Probabilidad: Baja (0.3)

Consecuencias: Catastrófico (1.0)

Descripción: Pérdida parcial o total de documentación, resultados experimentales y/o código fuente.

Contexto: En cualquier momento, debido a un error humano o un fallo catastrófico en el hardware o en el software, se pueden perder parte o la totalidad de los datos relativos al proyecto.

Análisis: La pérdida de los datos puede conllevar a grandes consecuencias, mayores cuanto más avanzado esté el proyecto y más próxima esté la fecha de finalización del proyecto, puesto que puede sea necesario volver a realizar la totalidad o parte el trabajo que se ha realizado hasta el momento.

Estrategia de resolución: Evasión y Protección.

Plan de acción: Para evitar la perdida de datos se usaran sistemas de control de versiones. Estos sistemas impiden que por error humano se borre un archivo o datos que nunca deberían ser borrados al poder recuperar en cualquier momento dicha versión del mismo. En caso de error de hardware, estas pérdidas se reducirán mediante el uso de copias de seguridad; tanto en discos físicos, servidores remotos o sistemas en la nube; que nos permitan recuperar versiones anteriores al fallo o por lo menos una versión lo más actualizada posible. Los sistemas utilizados se explicarán con más detalle en la sección 2.5.

R-8. Fallos del hardware

Probabilidad: Muy baja (0.2)

Consecuencias: Marginal (0.4)

Descripción: Una de las máquinas de experimentación sufre un fallo de hardware que requiere de una sustitución de componentes.

Contexto: Las máquinas de experimentación al igual que cualquier ordenador puede estropearse en cualquier momento pudiendo necesitar de piezas de repuesto. Estas piezas de repuesto no se pueden cambiar inmediatamente. Se necesita tiempo para que el encargado del mantenimiento pueda realizar las reparaciones, pudiendo tardar desde unas pocas horas a varios días. Durante este tiempo, la maquina no estará disponible para su uso.

Análisis: Las máquinas de experimentación son necesarias durante la fase de experimentación, no siendo necesarias durante el resto de fases. Estos experimentos se utilizarán para obtener los resultados experimentales, los cuales son importantes para el desarrollo del proyecto.

Estrategia de resolución: Evasión, Protección y Reserva.

Plan de acción: Utilizar para la experimentación máquinas dentro de un sistema de colas que seleccione automáticamente una de las máquinas disponibles. En caso de fallo, se puede utilizar otra de las máquinas disponibles para la experimentación. También es posible realizar otras tareas, estudiando la planificación, para dar tiempo a los técnicos a realizar las reparaciones.

R-9. Enfermedad del alumno

Probabilidad: Media (0.4)

Consecuencias: Crítico (0.8)

Descripción: El alumno sufre una enfermedad que le impide trabajar temporalmente en el proyecto.

Contexto: Las personas pueden enfermar especialmente en invierno. Durante el periodo de enfermedad no es posible realizar ningún trabajo relacionado con el proyecto. Dependiendo de la gravedad o la enfermedad esto se puede alargar por un único día a varias semanas.

Análisis: El proyecto no puede avanzar mientras el alumno está enfermo. Esto puede tener graves consecuencias en el avance del proyecto al causar retrasos en las diferentes actividades que conforman el proyecto.

Estrategia de resolución: Reserva.

Plan de acción: En caso de que se produzca el riesgo será necesario hacer un reajuste en la planificación empleando las posibles holguras. Además, se aumentará el tiempo dedicado al proyecto (horas extra) para recuperar el tiempo perdido, con su consiguiente aumento en el coste del proyecto.

2.4.3. Riesgos priorizados

Una vez que hemos identificado los riesgos se va a realizar una lista priorizada con ellos que servirá para ver cuales hay que tener más en cuenta. Esta lista se ordenará por la exposición al riesgo que es el producto de la probabilidad por las consecuencias (Exposicion = Probabilidad $*$ Consecuencias). Para que sea más sencillo de entender la exposición al riesgo se categoriza en los siguientes valores: bajo (<= 0.1), moderado (>0.1 - <= 0.2), significante (>0.2 - <= 0.3), alto (>0.3). En los riesgos que tengan varios valores en probabilidad y/o consecuencias se tendrá en cuenta el peor caso (el valor mayor) a la hora mostrar los datos y de ordenar. La lista priorizada de los datos se puede ver en la Cuadro 2.5.

Riesgo	Consecuencias	Probabilidad	Exposición
R-9. Enfermedad del alumno	Crítico (0.8)	Media (0.4)	Alto (0.32)
R-7. Pérdida de datos	Catastrófico (1.0)	Baja (0.3)	Significante (0.3)
R 1. Cambios en la propuesta de modelo	Crítico (0.7)	Baja (0.4)	Significante (0.28)
R-2. Cambios en los requisitos (Etapa Construcción)	Crítico (0.7)	Media (0.4)	Significante (0.28)
R 3. No completitud o ambigüedad de los requisitos.	Crítico (0.7)	Baja (0.4)	Significante (0.28)
R 5. Retraso en la finalización de actividades.	Crítico (0.5)	Media (0.5)	Significante (0.25)
R 4. Mal diseño. R-6. Errores en bibliotecas de terceros	Marginal (0.4) Marginal (0.3)	Media (0.3) Media (0.4)	Moderado (0.12) Moderado (0.12)
R-8. Fallos del hardware	Marginal (0.4)	Muy baja(0.1)	Bajo (0.04)

Cuadro 2.5: Lista priorizada de los riesgos

2.5. Gestión de configuraciones

El actual proyecto tiene como propósito el desarrollo de un modelo de programación. Por tanto dicho proyecto constará principalmente de artefactos relativos a la documentación, código fuente del prototipo y de los casos de estudio, y los resultados experimentales. A continuación hablaremos con más detalle de cómo se gestionarán los distintos tipos de artefactos.

2.5.1. Documentación

Toda la documentación, incluido el presente documento, se almacenará en un directorio en la nube usando el servicio de Internet llamado Dropbox y al finalizar el proyecto, también se almacenará en el CD-ROM que acompaña a este documento (ver apéndice A).

Dropbox es un servicio en la nube que te permite sincronizar de forma automática todo el contenido de un directorio de tu equipo con un directorio virtual en sus servidores. A este servicio pueden estar conectados múltiples dispositivos, lo cual nos permite acceder al contenido de la carpeta virtual desde cualquiera de los ordenadores en los que trabajemos, sincronizando la carpeta o accediendo a través de su aplicación web, o podemos acceder a través de dispositivos móviles mediante su app. Además Dropbox también nos ofrece un sistema sencillo de control de versiones, lo cual nos aporta seguridad en caso de borrado o modificación indebida de algún documento.

2.5.2. Código fuente del prototipo y de los casos de estudio

El código fuente del prototipo y de los casos de estudio se almacenará en un repositorio de código en una VPS (Virtual Private Server) usando la aplicación GitLab y al finalizar el proyecto, también se almacenará la última versión en el CD-ROM que acompaña a este documento (ver apéndice A). Dicha VPS estará almacenada en un servidor con discos redundantes al cual se le realizarán copias de seguridad semanales. La propia máquina virtual también sincroniza sus datos con otra VPS de mismas características.

GitLab es una aplicación web para la gestión de repositorios git. Además de repositorios git también tiene soporte para wiki y gestión de incidencias lo cual nos permite una gestión más completa de nuestros proyectos software. Por último, también es compatible con otras servicios de gestión de repositorios como GitHub, permitiéndonos importar proyectos que tuvieramos almacenados allí.

2.5.3. Resultados experimentales

Los resultados experimentales, son un caso especial de documentación formada por archivos de texto plano que contienen los resultados de los experimentos realizados con el prototipo. Una vez realizados los experimentos, estos archivos no se pueden modificar y se almacenarán en un archivo comprimido junto con la documentación del proyecto. En dicho archivo comprimido se indicará la máquina donde se ha realizado la experimentación y con qué versión del prototipo se ha probado.

Los resultados experimentales, al igual que la documentación, se almacenará en un directorio en la nube usando el servicio de Internet llamado Dropbox; también se almacenará en la máquina donde se realizaron las pruebas y al finalizar el proyecto, se almacenará la última versión en el CD-ROM que acompaña a este documento (ver apéndice A).

Capítulo 3

Seguimiento del proyecto

En el presente capítulo se describe cómo se ha realizado el trabajo de desarrollo del proyecto con respecto al plan previsto durante las diferentes iteraciones. Este seguimiento se realiza para poder detectar cualquier inconsistencia, en términos de tiempo, entre la planificación y las actividades que se están realmente realizando.

3.1. Seguimiento de las iteraciones

Para cada iteración se muestra la siguiente información:

- Desarrollo real: aquí se muestran las fechas de inicio y fin reales de cada actividad realizada junto con la diferencia en días entre la fecha de finalización planificada y la real.
- Riesgos producidos: se manifiestan los riesgos que se han producido y cómo se ha llevado a cabo el plan previsto.
- Observaciones: cualquier otro detalle observado durante la iteración.

3.1.1. Seguimiento primera iteración

Riesgos producidos

- R-9. Enfermedad alumno: durante la actividad de experimentación el alumno sufrió una enfermedad que le impidió trabajar durante 2 días. Sin embargo, esta actividad pudo completarse antes del tiempo esperado debido a que se disponía de tiempo sobrante. Este riesgo, que en su momento no se tuvo en cuenta, ha sido añadido a la gestión de riesgos.
- R-5. Retraso en la finalización de las actividades: En las actividades de *aprendizaje de CUDA*, *aprendizaje de Hitmap*, *desarrollo del modelo*, *análisis del modelo* y *diseño del prototipo*; sus fechas de finalización fueron posteriores a las planificadas. Debido a que los retrasos no fueron en fechas cercanas a la fecha planificada para el fin de la iteración y que no se consideraron graves, se optó por aceptar el riesgo.

Nº	Nombre de la actividad	Fecha inicio	Fecha fin	Retraso (días)
1	Inicio	2/03/15	2/03/15	0
2	Aprendizaje de CUDA	2/03/15	16/03/15	1
3	Aprendizaje de Hitmap	17/03/15	24/03/15	2
4	Exploración del estado del arte	25/03/15	28/03/15	-1
5	Propuesta de posibles modelos	29/03/15	3/04/15	0
6	Selección de modelo	4/04/15	4/04/15	0
7	Búsqueda de problemas de referencia	4/04/15	4/04/15	-1
8	Desarrollo del modelo	5/04/15	15/04/15	2
9	Análisis del prototipo	16/04/15	18/04/15	1
10	Diseño del prototipo	19/04/15	22/04/15	2
11	Construcción del prototipo	23/04/15	24/04/15	0
12	Construcción del experimento Jacobi: GPU vRef	25/04/15	26/04/15	-3
13	Construcción del experimento Jacobi: GPU v1	27/04/15	28/05/15	-4
14	Experimentación	29/05/15	2/05/15	-2
15	Planificación siguiente iteración	3/05/15	5/05/15	-1
16	Fin	7/05/15	7/05/15	-1

Cuadro 3.1: Seguimiento de la primera iteración

Observaciones

Durante esta iteración se ha descubierto que la planificación de actividad no ha estado muy ajustada, esto es debido a la inexperiencia del alumno a la hora de realizar planificaciones temporales. Por ejemplo, no se ha tenido en cuenta que en esta iteración solo se implementaba una interfaz con un mínimo de funcionalidad, por tanto el tiempo de implementación fue mínimo. Otro ejemplo, es que no se ha tenido en cuenta que las versiones de referencia de los experimentos se pueden obtener de fuentes públicas oficiales.

En esta iteración ha habido una serie de tareas que duraron más de lo estimado, como una planificación muy ajustada o por producirse un riesgo, en especial la tarea de desarrollo del modelo y la tarea de experimentación que fueron las más largas. La primera se produjo porque hubo ciertos detalles difíciles de concretar (véase rol de los parámetros y variables internas en el Capítulo 6). La segunda se debió a que el alumno sufrió de una enfermedad que le impidió trabajar durante 2 días.

3.1.2. Seguimiento segunda iteración

Riesgos producidos

- R-5. Retraso en la finalización de las actividades: En las actividades de *construcción del prototipo*, *construcción del experimento Jacobi: GPU v2* y *Experimentación*; sus fechas de finalización fueron posteriores a las planificadas. Debido a que los retrasos no fueron en fechas cercanas a la fecha planificada para el fin de la iteración y que no se consideraron graves, se optó por aceptar el riesgo.

Nº	Nombre de la actividad	Fecha inicio	Fecha fin	Retraso (días)
1	Inicio	7/05/15	7/05/15	0
2	Modificación del modelo	7/05/15	8/05/15	0
3	Análisis del prototipo	9/05/15	9/05/15	-1
4	Diseño del prototipo	10/05/15	13/05/15	-1
5	Construcción del prototipo	14/05/15	20/05/15	2
6	Construcción del experimento Jacobi: GPU v2	21/05/15	22/05/15	2
7	Experimentación	23/05/15	23/05/15	1
8	Redacción del artículo para las Jornadas de Paralelismo	24/05/15	1/05/15	-2
9	Revisión del artículo	2/06/15	7/06/15	0
10	Entrega del artículo	8/06/15	8/06/15	0
11	Planificación de la siguiente iteración	8/06/15	9/06/15	0
12	Fin	10/06/15	10/06/15	0

Cuadro 3.2: Seguimiento de la segunda iteración

Observaciones

En esta iteración los cambios en el modelo fueron mínimos, por este motivo la tarea modificación del modelo se realizó sin contratiempos y las tareas: análisis del modelo y diseño del prototipo; tardaron menos en terminarse de lo esperado. Sin embargo, la construcción del prototipo se alargó más debido a la necesidad de solventar algunas dificultades técnicas presentadas (ver rol de los parámetros en el Capítulo 9). Esto retrasó el resto de actividades, hasta que se recuperó el retraso debido a que la experimentación duró menos de lo esperado. En el caso de la redacción del artículo para el congreso de las Jornadas de paralelismo se terminó antes de lo planificado. Sin embargo, la inexperiencia en la escritura de artículos científicos obligó a estar revisando el artículo hasta el último día.

3.1.3. Seguimiento tercera iteración

Riesgos producidos

- R-4. Mal diseño: Durante la fase de *construcción del prototipo*, se detectó que el diseño estaba incompleto y que no contemplaba una serie de casos (ver CommCPU en el Capítulo 8 y el lanzamiento asíncrono de kernels en la sección 9.4.1). Esto obligó a modificar el diseño durante esta fase provocando un retraso grave en el desarrollo del proyecto.

- R-6. Error en bibliotecas de terceros: Durante la fase de construcción del prototipo, se detectó un error en una de las funcionalidades de la biblioteca. Dicha funcionalidad pese a no ser muy importante, simplificaba la implementación del prototipo. Dicho error se encontró y se solucionó implementando la misma funcionalidad en nuestro prototipo. Se dejó de usar la funcionalidad de la biblioteca de terceros y se informó al equipo de desarrollo de la biblioteca del error. Este riesgo que en su momento no se tuvo en cuenta ha sido añadido a la gestión de riesgos.

Nº	Nombre de la actividad	Fecha inicio	Fecha fin	Retraso (días)
1	Inicio	1/10/15	1/10/15	0
2	Modificación del modelo	1/10/15	4/10/15	0
3	Análisis del prototipo	5/10/15	5/10/15	-1
4	Diseño del prototipo	7/10/15	11/10/15	-1
5	Construcción del prototipo	12/10/15	20/10/15	3
6	Construcción del experimento Jacobi: CPU vRef	21/10/15	21/10/15	2
7	Construcción del experimento Jacobi: CPU v1	22/10/15	23/10/15	2
8	Construcción del experimento MatrixAdd: CPU y GPU vRef	24/10/15	25/10/15	3
9	Construcción del experimento MatrixAdd: CPU y GPU v1	26/10/15	27/10/15	3
10	Construcción del experimento Matrix-Mult: CPU y GPU vRef	28/10/15	28/10/15	2
11	Construcción del experimento Matrix-Mult: CPU y GPU v1	29/10/15	30/10/15	2
12	Experimentación	31/10/15	1/10/15	2
13	Redacción del artículo para HLPGPU	2/10/15	11/11/15	0
14	Revisión del articulo	12/11/15	15/11/15	0
15	Entrega del articulo	16/11/15	16/11/15	0
16	Fin	16/11/15	16/11/15	0

Cuadro 3.3: Seguimiento de la tercera iteración

- R-5. Retraso en la finalización de las actividades: Con motivo de que se tardó mucho en la construcción del prototipo hubo una gran cantidad de retrasos, entre 2 y 3 días después de lo planificado, en las actividades posteriores. Debido a ésto en la actividad de redacción del artículo se decidió aumentar el tiempo de actividad (horas extra) aumentando los costes del proyecto.

Observaciones

Durante esta fase en las primeras actividades no ha habido grandes retrasos, de hecho todo lo contrario, las actividades terminaron en menos tiempo de lo planificado. Sin embargo debido a una serie de riesgos que se produjeron en la actividad de construcción del prototipo, se produjo una serie de retrasos que obligó a aumentar el tiempo dedicado al proyecto durante la actividad de redacción del artículo. Este aumento del tiempo fue de 10 horas, 1 hora extra al día, y no fueron más debido a que la experiencia previa en la redacción de artículos permitió reducir el tiempo de realización (10 horas extra frente al retraso de 16 horas).

3.2. Conclusiones de la planificación

En general, el proyecto no tuvo grandes desajustes en cuanto a la planificación a excepción de los provocados por algunos riegos. También hay que añadir que al principio no se ajustaron bien algunas actividades, debido a la inexperiencia, y que algunos de los riesgos producidos al principio no se tuvieron en cuenta. Por último, hay que indicar que debido a un gran retraso en la última iteración, se debió aumentar las horas de trabajo dedicadas al proyecto teniendo como consecuencias un aumento de los costes del mismo. Este aumento de los costes fue del sueldo por 10 horas de trabajo aumentando los costes en 125 €. Con esto los costes finales ascienden a: **16.109 €**

Parte II

Conocimientos previos

Capítulo 4

Introducción a CUDA

CUDA [9, 12] (Compute Unified Device Architecture) es una arquitectura de cálculo paralelo masivo creada por la empresa NVIDIA que aprovecha la gran potencia de las GPUs (unidad de procesamiento gráfico) de sus tarjetas gráficas, para proporcionar un incremento extraordinario del rendimiento del sistema al delegar en ellas ciertas tareas. El hecho delegar tareas que no forman parte del procesamiento gráfico a la GPU es lo que se llama GPGPU (General-Purpose compute on GPUs).

En este capítulo se realizará una pequeña introducción al uso de dispositivos GPU de NVIDIA como dispositivo acelerador hardware. Con esta introducción, se pretende que el lector comprenda la complejidad de de la programación y algunos de los problemas que surgen durante el desarrollo de aplicaciones usando este tipo de dispositivos, aclarándole la motivación y objetivos de este proyecto.

Este capítulo está dividido en dos partes: la primera habla de la arquitectura de los dispositivos GPU de NVIDIA y la segunda sobre el modelo de programación que los hace posible funcionar.

4.1. Arquitectura de CUDA

Las tarjetas gráficas modernas están formadas por una o varias GPUs, las cuales son capaces de ejecutar una gran cantidad de hilos de forma simultánea. Estos hilos se organizan de forma jerárquica en la que un grupo de hilos forma un bloque de hilos y un grupo de bloques forma un grid. Se puede identificar cada hilo de forma individual mediante un ID formado por: el ID del hilo dentro del bloque y el ID del bloque dentro del grid (ver Figura 4.1). Estos dispositivos también disponen de una memoria propia donde almacenan los datos que puedan requerir las GPUs.

4.1.1. Funcionamiento de la GPU

Las GPUs modernas están formadas por un conjunto de multiprocesadores, llamados streaming multiprocessors (SMs), los cuales tienen la capacidad para ejecutar simultáneamente gran cantidad de hilos. Cuando se ejecuta un programa CUDA, los bloques del grid que lo forman se enumeran y se distribuyen entre los multiprocesadores disponibles; los que no dispongan de multiprocesador esperan para ser ejecutados. Los hilos de un bloque de hilos

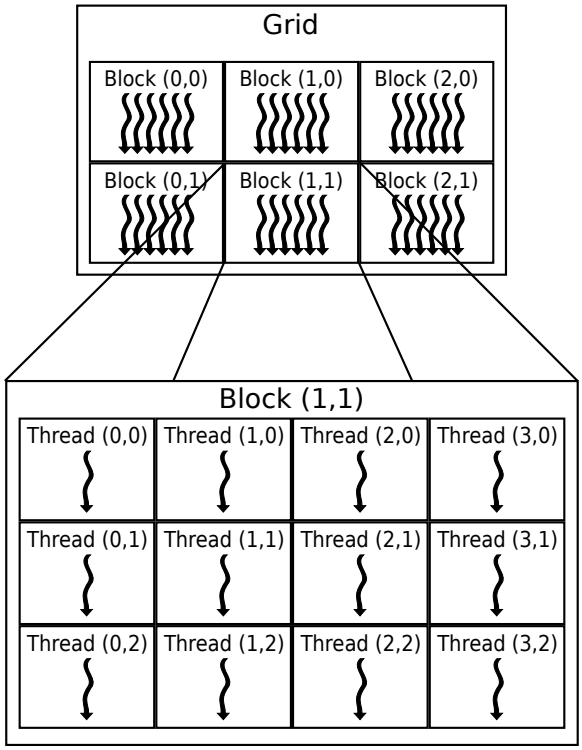

Figura 4.1: Jerarquia de un Grid.

son ejecutados concurrentemente por un mismo multiprocesador, y multiples bloques de hilos de un mismo grid pueden ser ejecutados concurrentemente en un mismo multiprocesador. Tan pronto como un bloque de hilos termina, otro nuevo bloque de hilos de entre los que están esperando se ejecuta en el multiprocesador disponible.

Un multiprocesador está diseñado para ejecutar cientos de hilos de forma simultánea. Para manejar tal cantidad de hilos, se emplea una arquitectura única llamada SIMT (Single-Instruction, Multiple-Thread).

Arquitectura SIMT

El multiprocesador crea, administra, planifica y ejecuta hilos en grupos de 32 hilos paralelos llamados warps. Los hilos individuales que componen el warp empiezan juntos en la misma dirección de memoria de programa. Ésto es porque un warp tienen un contador de programa único para todos los hilos, pero cada uno dispone de su propio registro de estado. Cada warp, y cada bloque, es libre para ramificarse y ejecutarse de forma independiente.

Cuando a un multiprocesador se le da uno o más bloques de hilos para ejecutar, este lo divide en warps y un planificador de warps planifica su ejecución. La forma en la que un bloque es dividido en warps siempre es la misma; cada warp contiene hilos consecutivos, incrementando el valor del identificador del hilo donde el primer warp contiene el hilo 0.

Los hilos de un warp ejecutan una misma instrucción en cada momento, por lo que la

máxima eficiencia se consigue cuando los 32 hilos de cada warp siguen la misma ruta de ejecución. Si algunos hilos de un warp divergen por una condición dependiente de algún dato, el warp ejecuta secuencialmente cada una de las bifurcaciones por los que tiene que pasar el programa, desactivando los hilos que no están en esa bifurcación. Cuando todos los caminos se han completado, los hilos vuelven a converger a la misma ruta de ejecución.

La arquitectura SIMT es parecida a la organización de vectores SIMD (Single Instruction, Multiple Data) donde una única instrucción controla múltiples elementos de procesamiento. La diferencia se encuentra en que la organización de vectores SIMD se realiza mediante software, mientras que las instrucciones SIMT especifican la ejecución y el comportamiento de ramificación en el flujo de programa de un único hilo. A diferencia de las máquinas de vectores SIMD, SIMT permite a los programadores escribir código paralelo independiente a nivel de hilo así como código de datos paralelos para hilos coordinados. Para crear un código más correcto, el programador puede ignorar esencialmente el comportamiento del SIMT; sin embargo, se pueden obtener mejoras sustanciales de rendimiento, si se tiene cuidado de que rara vez se requiera de una diversificación en el flujo de programa de los hilos.

4.1.2. Jerarquía de memoria

Las memorias a las que accede una GPU son jerárquicas donde cada hilo puede acceder a distintos niveles de memoria para obtener los datos de los que precisa (ver Figura 4.2).

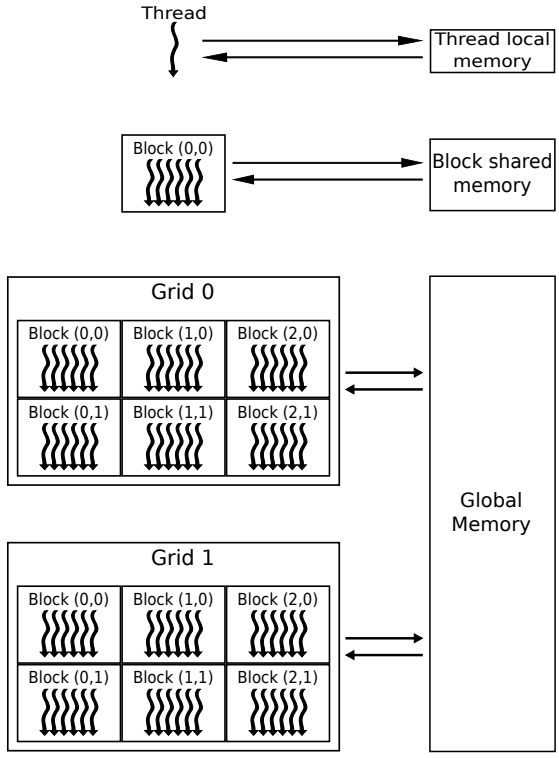

Figura 4.2: Jerarquia de memoria.

Cada dispositivo CUDA dispone de su propia memoria, llamada memoria global, que es compartida por las GPU de las que disponga el dispositivo y, por consiguiente, por todos los grids lanzados por en el dispositivo; además el tiempo de vida de los datos en esta memoria es el mismo que el tiempo de vida de la aplicación que lo usa. La memoria global es una memoria GDDR (Graphics Double Data Rate) DRAM. Esta memoria es ligeramente diferente de una DRAM utilizadas en una placa base en que la GDDR es principalmente una memoria intermedia de las imágenes usadas para gráficos. Para aplicaciones gráficas, esta memoria contiene imágenes de vídeo, y los datos de las texturas para renderizado de tres dimensiones (3D), lo cual requiere de un gran ancho de banda, lo cual produce unas mayores latencias que una memoria principal. Para computación paralela el gran ancho de banda compensa las bajas latencias.

Por debajo de la memoria global, dentro de cada multiprocesador, se dispone de una memoria mas pequeña y rápida llamada: memoria shared. Esta memoria, es compartida por todos los hilos del mismo bloque y sus datos tienen el mismo tiempo de vida que el bloque que la usa.

Por último, cada hilo dispone una serie de registro los cuales constituyen su memoria propia, donde se almacenan los datos propios de cada hilo. Los registros son las memorias más rápidas a las que puede acceder un hilo, sin embargo, el número de registros disponibles para cada hilo es muy limitado, por lo que normalmente se utilizan para aquellos datos a los que se accede con mayor frecuencia.

4.2. Modelo de programación

Para simplificar el aprendizaje de CUDA, se ha desarrollado una extensión del lenguaje C llamado CUDA C. Esta extensión del lenguaje permite al programador definir funciones en C, llamadas kernels, que cuando son llamadas, son ejecutadas N veces en paralelo por N hilos CUDA diferentes.

4.2.1. Kernels

Un kernel es definido usando el especificador de declaración `__global__` y durante la llamada, se le deben proporcionar el número de bloques junto con el número de hilos por bloque. Para ello se usa la nueva sintaxis para la configuración de ejecución <<< ... >>>. A cada hilo en ejecución se le da un identificador único que es accesible por el propio hilo a través de la variable incorporada `threadIdx`. En la Figura 4.3 se puede ver a suma de dos vectores A y B de tamaño N y el almacenamiento de su resultado en el vector C.

Los hilos de un bloque los hilos se pueden organizar en hasta 3 dimensiones (x, y, z) según las necesidades del programador. Normalmente para una mayor eficiencia se recomienda que el número de hilos de la dimensión x sea múltiplo de 32, el tamaño de un warp (ver sección 4.1.1). Con el grid ocurre lo mismo que con los bloques de hilos, también pueden organizarse en hasta 3 dimensiones (x, y, z). En la Figura 4.4 se puede ver a suma de dos matrices A y B de tamaño $N * N$ y el almacenamiento de su resultado en la matriz C.

4.2.2. Transferencias de memoria

Cuando se ejecuta un kernel de CUDA, los datos de los parámetros de llamada son transferidos al dispositivo para usarlos durante la ejecución. Previamente a ser transferidos los

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
 int i = threadIdx.x;
 C[i] = A[i] + B[i];
}

int main() {
 ...
 // Kernel invocation with N threads
 VecAdd<<<1, N>>>(A, B, C);
 ...
}
```

Figura 4.3: Ejemplo de definición de un kernel en CUDA.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
 int i = threadIdx.x;
 int j = threadIdx.y;
 C[i][j] = A[i][j] + B[i][j];
}

int main() {
 ...
 // Kernel invocation with one block of N * N * 1 threads
 int numBlocks = 1;
 dim3 threadsPerBlock(N, N);
 MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
 ...
}
```

Figura 4.4: Ejemplo de kernel que se ejecuta en un bloque bidimensional.

datos, la memoria en la que se van a alojar debe ser reservada. Sin embargo, CUDA solo puede hacer esto de forma automática si el tamaño se la memoria es estático, es decir, se conoce en tiempo de compilación. Cuando esto no es posible, la API de CUDA nos proporciona una serie de funciones que nos permiten realizar de forma manual todas las operaciones con la memoria que necesitemos.

Cuando se desean transferir datos de forma manual al dispositivo o al contrario, primero realizaremos la reserva de la memoria con la función `cudaMalloc`, a la cual proporcionaremos un puntero del tipo de los datos y el tamaño en bytes de la memoria que se desea reservar. La transferencia de memoria se realiza con la función `cudaMemcpy`, a la cual proporcionaremos el puntero a la zona de memoria de destino (host o dispositivo), el puntero a la zona de memoria de origen (host o dispositivo), el tamaño en bytes de la memoria que se va a transferir y una constante que indica si se hace la transferencia desde el dispositivo al host o al revés. Se puede ver un ejemplo en la Figura 4.5.

Cuando ya se han terminado de utilizar la memoria, al igual que con la reserva dinámica

en el host, hay que liberar los recursos utilizados; para ello, se utiliza la función `cudaFree`, a la que se le pasa un puntero a la memoria a liberar.

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N) {
 int i = blockDim.x * blockIdx.x + threadIdx.x;
 if (i < N)
  C[i] = A[i] + B[i];
}

// Host code
int main() {
 int N = ...;
 size_t size = N * sizeof(float);
 // Allocate input vectors h_A and h_B in host memory
 float* h_A = (float*)malloc(size);
 float* h_B = (float*)malloc(size);
 // Initialize input vectors
 ...
 // Allocate vectors in device memory
 float* d_A;
 cudaMalloc(&d_A, size);
 float* d_B;
 cudaMalloc(&d_B, size);
 float* d_C;
 cudaMalloc(&d_C, size);
 // Copy vectors from host memory to device memory
 cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
 cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
 // Invoke kernel
 int threadsPerBlock = 256;
 int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
 VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
 // Copy result from device memory to host memory
 // h_C contains the result in host memory
 cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
 // Free device memory
 cudaFree(d_A);
 cudaFree(d_B);
 cudaFree(d_C);
 // Free host memory
 ...
}
```

Figura 4.5: Ejemplo de programa CUDA con memoria dinámica.

Capítulo 5

Introducción a Hitmap

Hitmap es una biblioteca para tiling (o teselas) jerárquico y mapeo de arrays densos. Está basado en el modelo de programación distribuida SPMD (Single Program, Multiple Data), usando abstracciones para declarar estructuras de datos con una visión global, y automatiza la partición, el mapeo, y la comunicación de las jerarquías de tiles, sin dejar de ofrecer un buen rendimiento. Los tiles o teselas en Hitmap, es una estructura de datos para la representación avanzada de datos homogéneos mucho más flexible que los arrays. Esta es una representación que permite manejar de forma sencilla tanto matrices densas como matrices dispersas; además también permite mecanismos de selección avanzada, mediante el uso de strides (separaciones), y selecciones jerarquicas (subtiles o subtiles de subtiles).

En este capítulo, se va a describir la biblioteca Hitmap, la cual es usada en la implementación del prototipo. Este capítulo se divide en dos partes: en la primera se hablará de las características de Hitmap y la segunda en la que se hablará de la arquitectura.

5.1. Carácteristicas de Hitmap

Hitmap [6] tiene una serie de características para realizar tiling jerárquicos y el mapeo de arrays. Esta biblioteca está diseñada para simplificar el uso desde una vista local o global de la computación paralela, permitiendo la creación, manipulación y la comunicación eficiente de tiles y jerarquías de tiles.

En Hitmap, la distribución de los datos y las técnicas de balanceo de carga son módulos independientes que pertenecen a un sistema de plugins. Estas técnicas son llamadas desde el programa y aplicadas en tiempo de ejecución cuando son necesitados, usando información interna de la topología del sistema objetivo para distribuir los datos. El programador no necesita conocer el número de procesadores físicos de los que se dispone. En su lugar, se utilizan patrones de comunicación altamente abstractos para redistribuir los datos de los tiles.

La biblioteca Hitmap tiene soporte para:

- Generar la estructura de una topología virtual,
- Distribuir los datos entre diferentes procesadores de un topología mediante técnicas de balanceo de carga,

- Determinar automáticamente los procesadores inactivos en cualquier estado de computación,
- Identificar los procesos vecinos para usarlos en las comunicaciones,
- Crear patrones de comunicación para ser reutilizadas a lo largo de las iteraciones del algoritmo en el que sean necesarios.

Dichas funcionalidades están divididas en 3 categorías diferentes: Tiling, Mapeo y Comunicaciones.

5.1.1. Tiling arrays

Este módulo contiene una serie de funciones para la administración de los tiles jerárquicos y es la más importante para el prototipo de este proyecto. Las funciones de este modulo están pensadas para poder usarse de forma separada del resto de la biblioteca puesto que puede ser usada para mejorar el código secuencial, así como permitir una distribución manual de los datos para ser después ejecutados en paralelo.

Los tiles puede pueden ser definidos mediante un a partir del *dominio* de un array usando su rango particular de indices (ver matriz A en la Figura 5.1). Un tile puede ser derivado desde otro, especificando un *subdominio*, que es una subconjunto del rango de indices del tile padre. Se puede acceder a los elementos de la matriz original de dos sistemas de coordenadas diferentes: las coordenadas originales del array o el nuevo tile cuyas coordenadas empiezan en cero en todas sus dimensiones (ver las matrices B y C de la Figura 5.1). Los subdominios a su vez pueden contener *stride* transformando saltos regulares en los indices en una representación con coordenadas mucho más compactas (ver array D de la Figura 5.1).

La memoria utilizada para almacenar los elementos de los tiles se reserva bajo demanda. Los tiles pueden reservar memoria para almacenar sus propios elementos o pueden hacer referencia a los elementos de su antecesor. Los accesos a la memoria seleccionan de forma transparente al programador la memoria apropiada. Este sistema simplifica enormemente la partición de datos y la implementación de algoritmos paralelos, ya que nos permite crear un tile con un dominio definido y que la reserva de memoria se lleve a cabo en los subtiles definidos, permitiendo de forma sencilla la reserva de memoria distribuida. Además a esto hay que añadirle que hitmap soporta funciones de copia entre padres e hijos, tiles superpuestos, tiles extendidos, entre otros.

5.1.2. Mapeo

Hitmap, en su módulo de mapeo, realiza una separación clara entre: la topología y la distribución de los datos. Para ello implementa dos sistemas de plugins donde uno es para crear la topología virtual y el otro es para la distribución de los datos en dicha topología.

La topología virtual de Hitmap tiene como objetivo ocultar los detalles de la topología física al programador, numero de procesadores y distribución de los mismos, delegando a la biblioteca y en el correspondiente sistema de plugins la administración de la misma. Actualmente están desarrollados múltiples plugins para topologías diferentes: grids de procesadores de múltiples dimensiones o clusters de procesadores dependientes de la partición de datos entre otros.

La partición de los datos y su distribución se realiza de forma automática. Esto se realiza dividiendo los datos del array original en tiles dependiendo de la topología virtual elegida.

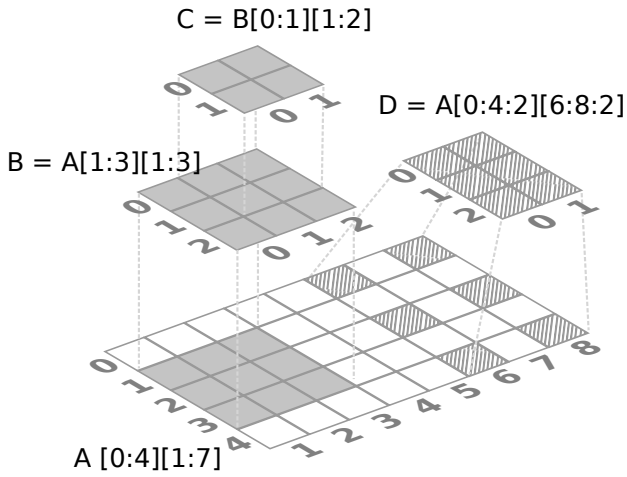

Figura 5.1: Ejemplo de selecciones jerárquicas de tiles en hitmap.

El modulo de partición recibe la topología virtual, el nombre de la función de distribución a aplicar y la estructura de datos a distribuir. Este devuelve un objeto que permite consultar la parte local del subdominio a cada procesador virtual, o puede ser usado para obtener los datos del subdominio asignado a otros procesadores virtuales.

5.1.3. Comunicaciones

Hitmap provee de una abstracción que permite la comunicación de los diversos fragmentos de datos distribuidos entre los distintos procesadores. El modulo de comunicaciones dispone de un amplio rango de abstracciones de comunicación incluyendo: comunicaciones punto a punto, intercambio de pares entre vecinos, desplazamientos entre los ejes de la topología virtual, comunicaciones colectivas, etc. Esta biblioteca apuesta por utilizar la información de los vecinos y los datos de los tiles creados de forma automática por el modulo de mapeado para que sin de forma transparente al programador las comunicaciones se adapten a la topología y a la partición de datos. Para conseguirlo, el programador le indica a la biblioteca los patrones de comunicación debe de utilizar. Dichos patrones están implementados junto con hitmap.

5.2. Arquitectura de Hitmap

Hitmap es una biblioteca que se ha desarrollado siguiendo una filosofía de programación orientada a objetos pero que ha sido implementada en lenguaje C. A continuación, se describe la arquitectura de la biblioteca describiendo primero como como están implementados los dominios de los tiles, lo cual tiene una alta relevancia en este proyecto, y posteriormente un pequeño repaso a como están implementados los 3 módulos.

5.2.1. Dominio de los tiles

En hitmap el dominio de los tiles está formado por una estructura de datos llamada *Shape*. Un shape representa un subespacio de los indices de un array definido como un paralelotopo n-dimensional regular, cuyos limites vienen determinados por n-*Signaturas* (ver fórmula 5.1). La cardinalidad de un shape, es decir la cantidad de elementos que contiene el shape, se obtiene con la formula 5.2, donde se puede ver que la cardinalidad es el producto de la cardinalidad de sus signaturas.

$$H \in Shape = (S_0, S_1, \ldots, S_{n-1}) \tag{5.1}$$

$$Card(H \in Shape) = \prod_{i=0}^{n-1} Card(S_i) \tag{5.2}$$

Una *Signatura* es una tupla de tres números enteros (ver fórmula 5.3) que representan los indices de uno de los ejes del dominio de tile. Estos números tienen el siguiente significado:

1. Begin: Donde se indica el primer indice en el eje del dominio del array original. Para que el primer indice del subtile coincida con el primer indice del array original, el valor de este número tiene que ser 0, en caso contrario habría un desplazamiento en las coordenadas del eje.

2. End: Donde donde se indica el ultimo indice en el eje del dominio del array original.

3. Stride: Es el salto regular entre los indices en el eje del dominio del array original. Para es caso de que no se desee realizar saltos en los indices el valor de este número debe ser 1.

Estas signaturas son muy flexibles y además obtener el indice original a partir de una posición del subtile es tan sencillo como aplicar la función lineal $f(x) = S.stride * x + S.begin$.

$$S \in Signature = (begin : end : stride) \tag{5.3}$$

$$Card(S \in Signature) = [(S.end - S.begin)/S.stride] \tag{5.4}$$

5.2.2. Tiles, Mapeo y Comunicaciones

Los tiles están implementados como estructuras de datos que contiene una referencia a los datos originales cuyo espacio de indices a sido definido por un Shape. En aquellos tiles que reservan su propia memoria, esta es la memoria a la que se hace referencia. En los subtiles, la memoria a la que se hace referencia es a la memoria de su padre, usando la información del shape para acceder a los datos de forma eficiente.

El módulo de topología y de distribución son interfaces al sistema de plugins los cuales son seleccionados por nombre en tiempo de ejecución. . El usar un sistema de plugins permite que los programadores puedan añadir sus propias técnicas a las que la biblioteca ya proporciona. Es sistema de plugins de la topología implementa funciones para distribuir los procesadores físicos en una topología virtual. Es sistema de plugins de distribución implementa funciones para distribuir un shape a lo largo de los procesadores de una topología virtual. Con el uso

de estos sistemas se obtiene una estructura layout que contiene información sobre el dominio local al procesador, la relación con sus vecinos y funciones para localizar otros subdominios.

Finalmente, la estructura de comunicaciones contiene información que de debe sincronizar o comunicar a los tiles en los procesadores. Con esta estructura interactúan funciones para construir diferentes esquemas de comunicación basados en los dominios de los tiles, la información de las estructuras layout y los patrones de comunicación con los vecinos necesitados. Los patrones son estructuras de datos que indican como debe actuar la biblioteca para una única comunicación. Internamente la biblioteca Hitmap esta implementada sobre la biblioteca de comunicaciones MPI, para ser portable a diferentes arquitecturas. Esta biblioteca explota múltiples técnicas de mejora del rendimiento en MPI, como los tipos derivados de MPI y las comunicaciones asíncronas.

Parte III

Análisis y Diseño

Capítulo 6

Modelo de comunicadores

En este capítulo, se pretende exponer de forma extensa la nueva solución ideada. Dicha solución es un nuevo modelo de programación al que se ha llamado *Modelo de comunicadores* y se ha desarrollado pensando en que cumpla con los objetivos marcados en este proyecto: Configuración automática de dispositivos, transferencias de memoria transparentes al programador y abstraer distintos tipos de programación heterogénea.

Este capítulo está dividido en tres partes. En la primera parte, se explicará el modelo general ideado que servirá para poder utilizar el mayor número posible de tipos de aceleradores. En la segunda parte, se hablará de cómo se adapta el modelo general para poder utilizar núcleos la CPU como si de un acelerador externo se tratase. En la última parte, se expondrá otro modelo ideado y los motivos por los cuales se descartó para este proyecto.

6.1. Modelo general

El modelo de comunicadores es un medio para simplificar la programación de aplicaciones que puede explotar plataformas de computación heterogénea que incluyen aceleradores y/o múltiples núcleos de CPUs. Este modelo se centra en una entidad llamada *comunicador*, la cual se encarga de servir como intermediario entre nuestro host y un dispositivo (ver Figura 6.1).

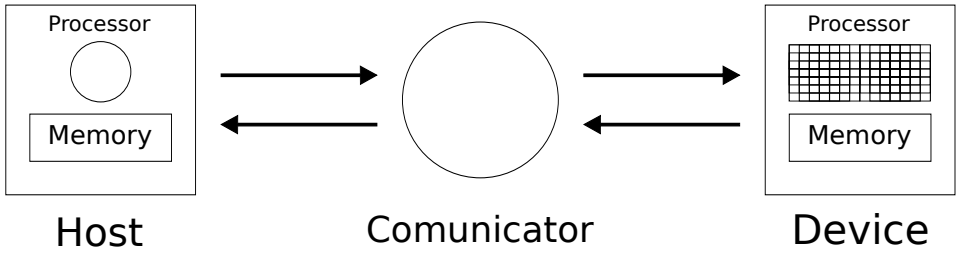

Figura 6.1: Visión general de un comunicador.

Un *comunicador* es una entidad que administra la ejecución de trabajos, llamados *Kernels*, junto con la memoria asociada en un dispositivo acelerador. Los *Kernels*, son funciones

adaptadas para ejecutarse en un tipo específico de acelerador que realizan operaciones concretas. Para poder administrar tanto la ejecución de kernels, como su memoria asociada, un comunicador debe gestionar automáticamente:

- La *configuración*: es la disposición en la que tienen que estar ciertas variables del dispositivo y parámetros de ejecución de los kernels para que éstos actúen de una manera determinada. Esto puede ser el número de hilos a ejecutar, configuración de memorias, etc.
- Las *comunicaciones*: son cada una de las transferencias de memoria que se realizan entre la memoria del host y la del acelerador. Estas transferencias necesitan realizarse antes de que se necesiten los datos que se transmiten.
- El *lanzamiento de kernels*: es el proceso por el que se prepara al acelerador para la ejecución de un kernel.

6.1.1. Configuración de los trabajos

En el modelo, los kernels necesitan indicar dos tipos de información que permiten al comunicador preparar su ejecución. Esta información se le indicará junto al código del kernel, debido a que esta información es independiente de los valores de los datos de entrada, es decir, es conocida en tiempo de implementación. Esta información consiste en:

- La caracterización del código del kernel: es un método para indicarle al comunicador parámetros característicos del código. Esto permite al comunicador obtener los valores óptimos, o cercanos al óptimo, para la configuración del dispositivo y los parámetros de lanzamiento del kernel. Está caracterización se indicará junto con el propio código del kernel (ver Figura 6.2 linea 2).
- El rol de los parámetros: es un método para indicar el uso que se le dará a los parámetros del kernel dentro del mismo (ver Figura 6.2 linea 4). Con esta información el comunicador puede controlar que transferencias entre las diferentes memorias necesita realizar en cada momento. Las posibilidades son:
  - Entrada de datos (IN): el kernel sólo leerá los datos de este parámetro, por tanto, los datos deben estar disponibles antes de su ejecución,
  - Salida de datos (OUT): el kernel sólo escribirá datos en este parámetro, por lo que sus datos tendrán que ser transferidos después de la ejecución cuando sea necesario,
  - Entrada y salida de datos (IO): el kernel leerá y escribirá datos en este parámetro, por lo cual, sus datos deben estar disponibles antes de su ejecución y si es necesario se deberán transferir después.

6.1.2. Comunicaciones

Los aceleradores disponen de su propio espacio de memoria, lo cual obliga a que se transfieran los datos que van a ser procesados, y los resultados obtenidos, entre la memoria del dispositivo acelerador y el host. La administración manual de estas transferencias es tediosa

```
1 /* Kernel characterizations */
2 KERNEL_CHAR(kernel, 1, full, low, high)
3 /* Kernel codes */
4 KERNEL(kernel, 2, OUT, HitTile_float*, var_dst,
5                   IN,  HitTile_float*, var_src){
6         /* code of kernel */
7 }
```

Figura 6.2: Ejemplo de caracterización.

y propensa a errores, lo cual aumenta el coste de desarrollo. Además, el uso de técnicas más avanzadas de transferencia de memoria requieren de un conocimiento preciso de la secuencia de lanzamiento de los kernels, complicando aún más la tarea. Por ejemplo, el solapamiento de la computación con comunicación donde los datos que se transfieren no pueden ser los mismos que se están usando en el kernel que se está ejecutando.

Un comunicador se asocia a un dispositivo en el momento de su creación. El dispositivo es un acelerador concreto. El comunicador administra además las imágenes de las variables en el espacio de memoria del dispositivo. El comunicador, puede decidir cuándo y cómo se producen las transferencias de memoria, dependiendo de su uso en los correspondientes kernels que están en cola para ejecutarse.

El modelo también permite al programador usar las variables, o estructuras de datos, originales en vez de definir unas nuevas (variables) que correspondan con las imágenes en el dispositivo acelerador. Las variables usadas por los comunicadores pueden ser de dos tipos diferentes según sus características: variables enlazadas y variables internas.

Variables Enlazadas

Las variables enlazadas, son variables del host que tienen imagen en el espacio de memoria del dispositivo acelerador (ver Figura 6.3).

El modelo permite enlazar una variable del host al comunicador. A partir de ese momento, se comvierte en una variable enlazada y sus datos no deben ser modificados por el programa del host hasta que se le aplique una operación de desenlazado. El primer kernel que requiera el uso de una variable enlazada como entrada (con un rol IN o IO) obliga al comunicador a asegurar de forma transparente que sus datos han sido transferidos. La memoria en el dispositivo necesaria para la imagen de la variable, puede ser reservada durante el enlazado o antes de la transferencia (dependiendo de la implementación).

La aplicación de una operación de desenlazado a una variable enlazada obligará a la transferencia de sus datos en el acelerador hacia el host si se trata de una variable de salida (con un rol OUT o IO). El programa principal esperará a que se termine la ejecución del kernels que usan la variable y la transferencia de los datos.

Variables Internas

Variables internas son variables cuyo alcance se limita al código ejecutado en el acelerador. Sólo se manejan dentro del espacio de memoria del dispositivo, y no tendrá asignación en el espacio de memoria del host. Por lo tanto, no se van transferir datos a la memoria del dispositivo (ver Figura 6.4).

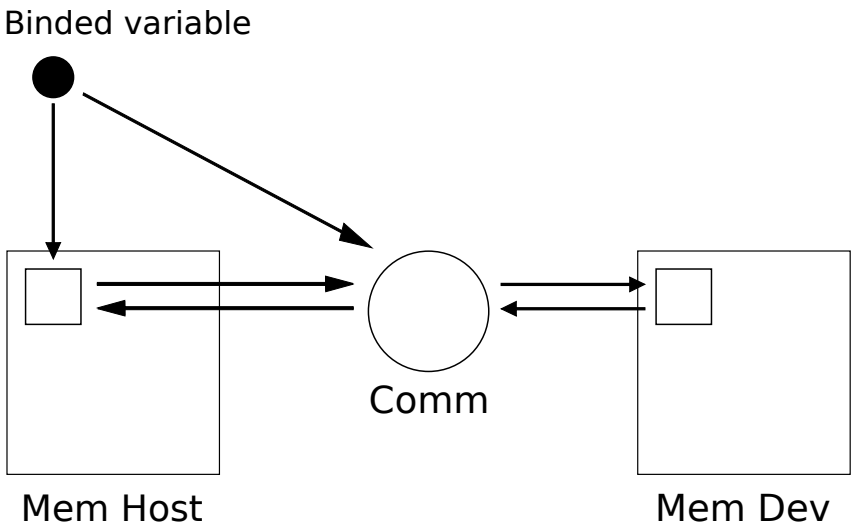

Figura 6.3: Variable enlazada.

Estas variables, se crean en el comunicador a través de una operación aplicada a una imagen de variable del sistema principal. El programador declara una estructura de datos sin asignación de memoria en el host. Esto se hace sólo para clonar el tipo, tamaño, y la estructura en el espacio de memoria del comunicador. Desde el momento de creación de la variable interna, la imagen de esta variable podría ser utilizada por los kernels que se ejecuten en este comunicador.

Para eliminar una variable interna es necesario aplicar otra operación utilizando la variable de referencia. Una vez aplicada la operación de eliminación, el comunicador eliminará la variable en cuanto no sea necesaria, asegurándose que no se elimina antes de tiempo.

6.1.3. Lanzamiento de kernels

El lanzamiento de un kernel es el proceso por el cual se le indica al acelerador que debe iniciar en cuanto pueda la ejecución de un kernel. Este proceso se realiza a través de un planificador que organiza, a través de ciertas políticas, el orden de ejecución de los kernels en cola, para poder explotar técnicas avanzadas de computación con aceleradores (solapar comunicación con computación, ejecución de kernels concurrentes, etc.). Antes de iniciar el lanzamiento de un kernel, el planificador debe asegurar que este dispone de los datos que necesita (ver sección 6.1.2). Durante el lanzamiento, para obtener la configuración del dispositivo y los parámetros de lanzamiento del kernel se utiliza una caracterización del kernel (ver sección 6.1.1). El programador suministrará además el número de hilos que se quiere lanzar.

En la Figura 6.5 se puede ver el funcionamiento de un comunicador. Este comunicador dispone de una lista en la que encolan los kernels, los cuales tienen una serie de variables vinculadas. Esta lista de trabajos está administrada por el planificador del comunicador.

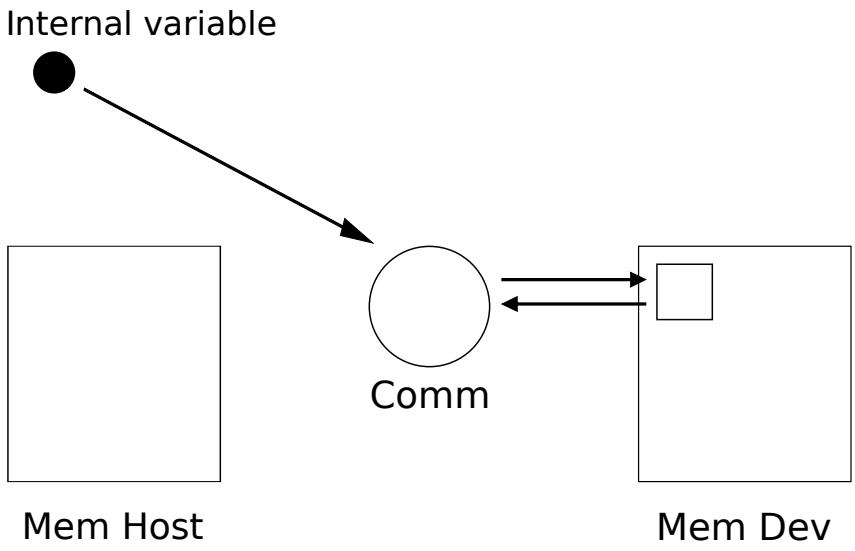

Figura 6.4: Variable interna.

6.2. CPU como acelerador

Con motivo de simplificar el uso de los comunicadores en entornos de computación heterogénea, se le ha añadido un caso particular de sistema de computación que no existe como dispositivo hardware externo, los núcleos de la CPU. Con este caso particular, se pretende acercar la programación en CPU a una más cercana a la de un acelerador hardware, permitiendo usar un mismo sistema, o abstracción de programación, para todo. Para que la CPU actúe como un acelerador se ha tomado como base el modelo de programación de GPU donde un kernel se ejecuta en muchos hilos y estos manipulan un conjunto de datos. Esto es una visión contraria a OpenACC [3] donde se desarrolla un algoritmo pensado para ejecutar en CPU, el cual posteriormente se transforma en una versión para el dispositivo objetivo.

6.2.1. Comunicaciones

Debido a que la CPU es parte del host, el hecho de realizar transferencias de memoria no tiene ningún sentido. Sin embargo, el programador debe tener la sensación de programar para un dispositivo externo, por tanto, el comunicador simula el uso de las variables que realiza un comunicador genérico. Para ello, se han simulado tanto las variables enlazadas como las variables internas, aunque cada una tiene sus particularidades.

Variables Enlazadas

Las variables enlazadas de CPU usan la propia variable en el host para su uso en la computación. Sin embargo, al igual que con los dispositivos externo, es necesario que la operación de desenlazado bloquee la ejecución del programa principal hasta que la variable esté disponible para su uso, es decir, que terminen todos los kernels en cola que requieran de esa variable. Esto convierte a las variables enlazadas, al igual que con los dispositivos

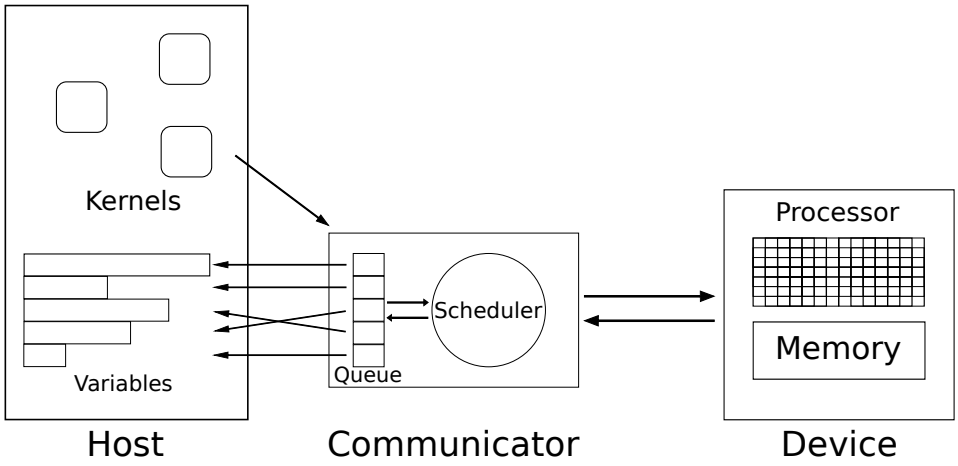

Figura 6.5: Comunicador detallado.

externos, en mecanismos de sincronización entre el comunicador y el programa principal.

Variables Internas

Las variables internas de CPU son estructuras de datos que aunque tienen toda su estructura interna definida, su memoria no ha sido reservada. Por este motivo, la operación de creación del comunicador de CPU debe realizar la reserva de memoria. En cambio, en la operación de eliminación de la variable interna el comunicador espera a que esta variable ya no sea necesaria y después libera la memoria reservada. Debido a que la memoria es reservada y liberada por el comunicador, las variables internas de CPU son variables cuya memoria está administrada por éste.

6.2.2. Lanzamiento de kernels

El lanzamiento de un kernel de CPU es un caso particular puesto que el comunicador debe emular la ejecución de una gran cantidad de hilos con un número limitado de núcleos. Para lograr esto, el comunicador debe ejecutar en cada hilo de CPU múltiples veces el mismo kernel hasta que el número total de veces que se ha ejecutado sea el necesario.

Al igual que en el caso de un comunicador para un dispositivo externo, el comunicador debe asegurarse que dispone de los datos antes de la ejecución del kernel (ver sección. 6.2.1).

6.3. Contextos: otra posible alternativa

Además de los comunicadores se propuso otro posible modelo que fue desestimado, es te modelo fue el modelo de contextos. Un *contexto* es una entidad encargada de la gestión completa de la ejecución de uno o varios algoritmos en uno o varios aceleradores hardware. Esta entidad se basa en el concepto de que un algoritmo es cualquier procedimiento computacional bien definido que toma un valor, o una conjunto de valores, como entrada y produce un valor, o un conjunto de valores, como salida [21].

A la hora de usar un contexto, por cada algoritmo que se desee ejecutar, se le deben facilitar los datos de entrada y el lugar donde debe colocar los datos de salida. A su vez, se le deben asignar los dispositivos en los que se desea que se ejecute el algoritmo. Posteriormente, se le debe indicar el flujo de trabajo entre los distintos *kernels*, funciones que realizan operaciones concretas y están adaptadas a su uso en aceleradores, que conforman el algoritmo.

A partir de todo lo indicado, el contexto se encarga de decidir: el orden de ejecución de los kernels, el dispositivo en el que se ejecutaran y cuando realizar las transferencias de memoria.

6.3.1. Posible ejemplo

```
1  Var a, b, c, d; // Variables locales a usar
2  Context con;
3  Create(con);     // Se crea el contexto.
4  AddDevice(con, CPU, 0); // Se asigna el dispositivo de CPU 0.
5  AddDevice(con, GPU, 0); // Se asigna el dispositivo de GPU 0.
6  AddInput(con, a); // Se indica que "a" es una entrada.
7  AddInput(con, b); // Se indica que "b" es una entrada.
8  AddOutput(con, c); // Se indica que "c" es una salida.
9  AddOutput(con, d); // Se indica que "d" es una salida.
10
11 WorkA(con, a, c); // Se indica que de debe ejecutar el
12                                    // trabajo "A"
13 WorkB(con, b, c); // Se indica que de debe ejecutar el
14                                    // trabajo "B"
15 WorkC(con, a, b, d); // Se indica que de debe ejecutar el
16                                          // trabajo "C"
17 Launch(con); // Lanza los trabajos. Esto se ejecuta de forma sincrona.
18
19 /* Se pueden manipular las variables c y d. */
20
21 Destroy(con); // Se elimina el contexto.
```

Figura 6.6: Posible ejemplo de la propuesta de contexto.

Como es posible ver en la Figura 6.6 los contextos dispondrían de sentencias de creación y destrucción (ver líneas 3 y 21). Después de crear el contexto se le deben asignar los dispositivos (ver líneas 4 y 5) y las variables de entrada (ver lineas 6 y 7) y las de salida (ver lineas 8 y 9). Posteriormente indicar los trabajos que se van a ejecutar (ver líneas 11, 13 y 15). La orden para que se ejecute (ver línea 17) podría ser sincrona, puesto que encierra los mecanismos de sincronización y transferencias.

En este ejemplo, los trabajos “A” y “B” se ejecutarán simultáneamente con “C” debido a que no tienen las mismas variables de salida y ninguna de las variables de entrada se ve alterada. En este caso, el contexto podría ejecutar “A” y “B” en un dispositivo y “C” en otro.

6.3.2. Desestimación de la alternativa

El motivo por el que se desestiman los contextos, es debido a que tendría un tiempo de desarrollo mayor a lo planificado por su alta complejidad. Esto se debe a que además

de desarrollar los objetivos de este proyecto, se deberían añadir políticas de planificación, selección de dispositivo y balanceo de carga.

Esta alternativa podría ser explorada como trabajo futuro, dónde se podrían utilizar los comunicadores como mecanismo de gestión del dispositivo a un nivel inferior a los contextos.

Capítulo 7

Análisis

En este capítulo se va a realizar el análisis previo al diseño del prototipo. Este análisis consistirá primero en obtener los requisitos del sistema, se obtendrán los casos de uso del mismo y por último se realizará un modelo de objetos que participan en el sistema.

7.1. Análisis de requisitos

En esta sección se describen los requisitos del sistema obtenidos a partir del modelo teórico planteado en el capítulo 6. Los requisitos del sistema, según Sommerville [18], son la descripción de los servicios proporcionados por el sistema y sus restricciones operativas. Estos requisitos reflejan las necesidades de los clientes de un sistema que ayude a resolver algún problema como el control de un dispositivo, hacer un pedido o encontrar información. Los requisitos se clasifican como:

- Requisitos funcionales: Son declaraciones de los servicios que debe proporcionar el sistema, la manera en que éste debe reaccionar a entradas particulares y cómo se debe comportar en situaciones particulares. También pueden declarar lo que el sistema no debe hacer.
- Requisitos no funcionales: Son restricciones de los servicios o funciones ofrecidas por el sistema. Incluyen restricciones de tiempo, proceso de desarrollo y estándares que se deben seguir. Los requisitos no funcionales a menudo se aplican al sistema en su totalidad.
- Requisitos del dominio: Son requerimientos que provienen del dominio de aplicación del sistema y que reflejan las características y restricciones de ese dominio. Pueden ser funcionales o no funcionales.

7.1.1. Requisitos funcionales

Ahora se procederá a realizar una descripción de los requisitos funcionales detectados a partir de analizar el modelo teórico. Como ya se describió anteriormente los requisitos funcionales son declaraciones de los servicios que debe proporcionar el sistema.

Los requisitos funcionales detectados, se pueden ver resumidos en el Cuadro 7.1.

Requisito	Descripción
RF-1	Enlazar variables.
RF-2	Desenlazar variables.
RF-3	Crear variable interna.
RF-4	Destruir variable interna.
RF-5	Comunicaciones automáticas.
RF-6	Caracterización del código del kernel.
RF-7	Rol en los parámetros del kernel.
RF-8	Configuración automática de los parámetros del kernel.
RF-9	Lanzamiento asíncrono de kernels.

Cuadro 7.1: Lista de requisitos funcionales

RF-1. Enlazar variables

El sistema tiene que poder enlazar una serie de variables al comunicador. Estas variables una vez enlazadas pasarán a ser llamadas variables enlazadas y serán administradas por el comunicador, el cual se encargará de la reserva de memoria en el acelerador, hasta el momento en el que el host desee desenlazarlas, cuando se liberarán los recursos.

RF-2. Desenlazar variables

El sistema debe permitir desenlazar las variables enlazadas. En el momento del desenlace, se debe el sistema debe esperar a que el uso de dicha variable haya terminado y los datos estén listos para ser usados. Este requisito corresponde a las variables enlazadas del modelo(ver sección 6.1.2).

RF-3. Crear variable interna

El sistema tiene que poder administrar variables al que se le haya indicado un dominio pero que no se le haya reservado memoria. A estas variables, a las que se les pasará a llamar variables internas, se las debe preparar pasa su uso, reservando la memoria necesaria, con los kernels que se ejecuten en el comunicador que los administra.

RF-4. Destruir variable interna

El sistema debe permitir eliminar variables internas. Cuando se desee eliminar dichas variables, el sistema se hará responsable del proceso de liberar los recursos. Este requisito corresponde a las variables internas del modelo(ver sección 6.1.2).

RF-5. Comunicaciones automáticas

El sistema debe administrar de forma automática las comunicaciones entre el host y el comunicador de forma que los datos siempre estén donde se les necesite. Para ello el sistema deberá utilizar la información proporcionada por el programa principal, como las variables enlazadas y variables internas por ejemplo.

RF-6. Caracterización del código del kernel

El sistema debe proporcionar un mecanismo que permita indicar al comunicador parámetros indicativos de ciertas características del código del propio kernel. Estos parámetros se utilizan asegurar que el funcionamiento durante la ejecución del kernel al que pertenecen sea lo más óptimo posible (ver RF-8).

RF-7. Rol en los parámetros del kernel

El sistema debe proporcionar un mecanismo que permita indicar el uso que se le va a dar a los parámetros que recibe el kernel. Se debe de poder indicar si los parámetros son usados para: entrada, salida o entrada y salida.

RF-8. Configuración automática de los parámetros del kernel

El sistema debe configurar de forma automática el dispositivo acelerador de forma que cuando se le ordene la ejecución de un kernel el dispositivo esté preparado para ejecutarlo de forma eficiente. Esta configuración se obtendrá a partir de información proporcionada mediante otras funcionalidades y datos del dispositivo y un modelo de selección de parámetros.

RF-9. Lanzamiento asíncrono de kernels

El sistema debe ejecutar los kernels cuando considere que es el mejor momento para su ejecución. Para ello el sistema debe de proporcionar un mecanismo de planificación, por sencillo que sea, y un sistema para que se puedan seguir enviando kernels para su ejecución sin que para ello detenga el programa principal que lo usa.

7.1.2. Requisitos no funcionales

A continuación se va proceder a realizar una descripción de los requisitos no funcionales detectados, los cuales se pueden ver resumidos en el Cuadro 7.2. Estos se han obtenido a partir del modelo teórico y de las restricciones impuestas para este proyecto.

Requisito	Descripción
RNF-1	Desarrollo en lenguaje C.
RNF-2	Desarrollo utilizando la biblioteca Hitmap.
RNF-3	Comunicadores compatibles con CUDA.
RNF-4	Comunicadores compatibles con CPU.

Cuadro 7.2: Lista de requisitos no funcionales

RNF-1. Desarrollo en lenguaje C

Para nuestro proyecto se ha definido como restricción que debe de programarse el prototipo en lenguaje C. Este lenguaje de programación, junto con C++ y Fortran, es uno de los lenguajes más utilizados en la programación de aplicaciones para computación de alto rendimiento. Estos lenguajes destacan por el hecho de que permiten un uso muy eficiente de los recursos, por desgracia a costa de dejarle la administración al programador, y son muy flexibles.

RNF-2. Desarrollo utilizando la biblioteca Hitmap

Para nuestro proyecto se ha definido como restricción que debe ser programada utilizando la biblioteca Hitmap. Utilizando Hitmap es fácil realizar una partición de datos en tiles y repartirlos en múltiples máquinas para realizar la computación conveniente. Al utilizar nuestro proyecto junto con esta biblioteca podría ser posible en el futuro programar aplicaciones para computación heterogénea distribuida.

RNF-3. Comunicadores compatibles con CUDA

Para poder probar el funcionamiento del modelo en aceleradores hardware, especialmente en GPUs, el prototipo tiene que ser compatible en los sistemas de ejecución en dispositivos aceleradores concretos. Los dispositivos GPU de NVIDIA son unos de los más utilizados en supercomputación por lo que para que sea útil nuestro prototipo debe poder demostrar que el modelo es útil para este tipo de dispositivo. Además a estos dispositivos se debe acceder usando el lenguaje CUDA puesto que nos permite optimizar la ejecución de formas que otros sistemas no permiten. Por ejemplo, manipulando la configuración de la caché L1.

RNF-4. Comunicadores compatibles con CPU

Entre los objetivos del proyecto figura unificar distintos paradigmas de programación heterogénea. Debido a la importancia de las CPUs en cualquier tipo de computación y que ciertos tipos de aceleradores utilizan el mismo paradigma, se ha decidido tratar a los núcleos de la CPU como un acelerador externo. De esta forma, uniendo los paradigmas a través de la abstracción de los comunicadores, puede ser posible facilitar aún más la computación heterogénea.

7.2. Casos de uso

En esta sección, se pretende realizar la especificación de los casos de uso de nuestro sistema. Según Pressman [16], un caso de uso narra una historia estilizada sobre cómo interactúa un usuario final (que tiene cierto número de roles posibles) con el sistema en circunstancias específicas. Aquellas entidades que interactúan con el sistema en esta historia son los llamados *Actores*. Los actores son las distintas personas (o dispositivos) que usan el sistema o producto en el contexto de la función y comportamiento que va a describirse.

Para el caso de nuestra biblioteca, tan solo habrá un caso de uso debido a que nuestro modelo unifica el uso de los aceleradores en una entidad única, cumpliendo así uno de los objetivos marcados (ver sección 1.2). En este caso de uso el actor principal será el programa principal, el cual es el que interactúa con el sistema, nuestro prototipo, y recibe sus resultados. Además tiene un actor secundario el cual será el acelerador del cual el comunicador hace de intermediario con el host.

Caso de uso: Lanzamiento de kernels.

Actor principal: Programa principal.

Objetivo en contexto: El objetivo es ejecutar uno o varios kernel en el acelerador obteniendo el resultado final de todas las ejecuciones.

Pre-condiciones: Ninguna.

Disparador: El programa necesita realizar un calculo que decide que es mejor ejecutarlo en un acelerador.

Escenario:

1. El programa principal: Crea un comunicador y lo enlaza a un dispositivo acelerador.
2. El programa principal: Enlaza las variables que contienen los datos con los que desea trabajar y donde desea recibir los resultados.
3. El programa principal: Crea las variables internas con las que trabajará el acelerador.
4. El programa principal: Lanza los trabajos que se deben ejecutar indicando las variables a usar.
5. El programa principal: Desenlaza las variables enlazadas y espera a que se desenlacen.
6. El programa principal: Elimina las variables internas del comunicador.
7. El programa principal: Elimina el propio comunicador.

Excepciones:

1. Alguna de las variables utilizadas para el lanzamiento de un kernel no está ni enlazada ni es interna: se avisa del error y no se lanza el kernel.
2. Se intenta desenlazar una variable no enlazada: se avisa del error y se cancela la acción.
3. Se intenta eliminar una variable que no es interna: se avisa del error y se cancela la acción.
4. Se intenta eliminar un comunicador no inicializado: se avisa del error y se cancela la acción.

Prioridad: Esencial

Actores secundarios: Acelerador hardware

7.3. Modelo de objetos

En esta sección se detallarán el modelo de los objetos obtenidos a partir del modelo teórico para el prototipo. Los modelos de objetos o de clases [16, 18] es una aproximación orientada a objetos donde se representan tanto los datos del sistema, como las operaciones que los manipulan, las relaciones entre los objetos y las colaboraciones. El modelo de objetos obtenido de puede observar en la Figura 7.1.

El modelo de objetos del proyecto está formado la clase Comunicador y sus especializaciones, Comunicador para GPU y Comunicador para CPU, y la clase Variable la cual representa tanto a las variables enlazadas como las internas.

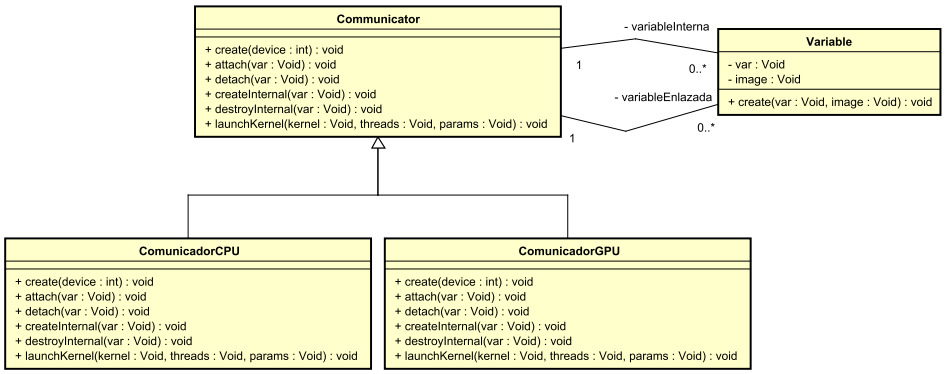

Figura 7.1: Modelo de objetos de los comunicadores.

La clase *Variable* está formada por la variable a enlazar y por su imagen en el dispositivo. En el caso de las variables enlazadas ambos apuntaran a variables con memoria propia, pero en caso de las variables internas una apuntará a la variables con memoria del dispositivo (image) y la otra a la variable sin memoria del host (var).

La clase *Communicator* contiene las funciones de enlazado de variables (attach), desenlazado de variables (detach), creación de variable internas (createInternal), destruccion de variables internas (destroyInternal) y para el lanzamiento de kernels (launchKernel).

Capítulo 8

Modelo de diseño

En este capítulo se a explicar el diseño del prototipo experimental creado a partir del modelo explicado previamente.

8.1. Modelo estructural

En esta sección hablaremos de la estructura que tiene nuestro prototipo ya sabiendo que se desarrollará en lenguaje C y utilizando la biblioteca Hitmap. Las estructuras de esta biblioteca junto con las funciones que las utilizan se pueden observar en la Fig. 8.1. Estas estructuras y funciones se detallan más adelante. Hay que tener en cuenta que todas las funciones pertenecientes a una estructura, aunque no está detallado en los diagramas, recibe como primer parámetro un puntero a la misma.

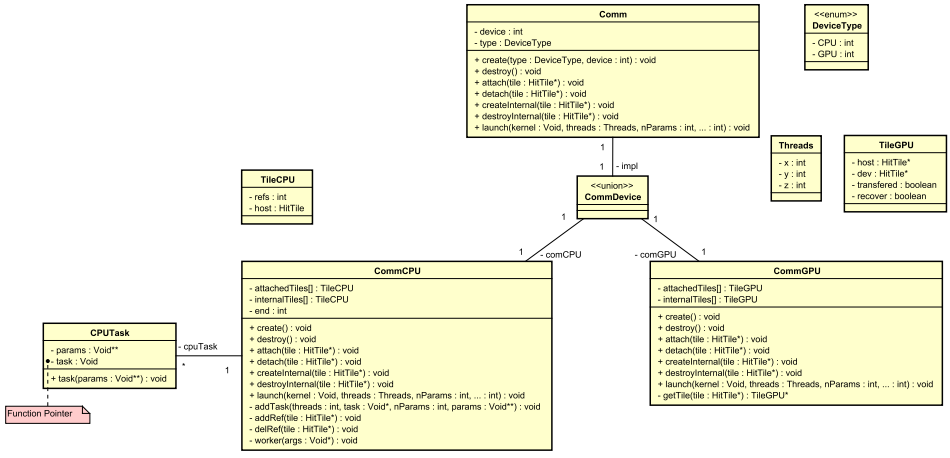

Figura 8.1: Clases que conforman los comunicadores.

8.1.1. Estructura Comm

La estructura *Comm*, junto con las funciones que la utilizan, se encargan de gestionar un comunicador, controla todo lo relativo a la ejecución de las tareas, enlazado de variables y creación de variables internas. Para nuestro prototipo las funciones que usan esta estructura, son las únicas que debe usar directamente el programador que utilice la biblioteca.

Este comunicador es genérico, por tanto se puede utilizar para CPU o GPU según le indique el usuario en la función de creación. Esto se consigue ya que dispone de una unión de las estructuras comunicador específicas de cada tipo de dispositivo, por tanto un comunicador solo podrá controlar el dispositivo que se le indique durante la creación. El motivo para usar una unión, es que al solo poder utilizar uno de las dos estructuras el usar una unión es mucho más eficiente en términos de memoria. Esto se debe a que las uniones hacen que sus datos compartan el mismo espacio de memoria al superponerlos, de forma que el espacio ocupado es el de la variable más grande.

Función create

La función *create* se encarga de inicializar, hacer las reservas de memoria y configurar todo lo necesario para el funcionamiento de los comunicadores. Esta función recibe como parámetros el tipo de dispositivo en el que se desea usar, y el numero de dispositivo. El tipo de dispositivo tiene que ser un valor del la enumeración *DeviceType*. El numero de dispositivo es un valor numérico de dispositivo que le asigna a cada dispositivo del mismo tipo (En nuestro prototipo la CPU solo puede recibir 0).

Función destroy

La función *destroy*, que es la contraparte de la función de creación, se encarga de liberar todos los recursos asociados al comunicador, así como de desenlazar las variables enlazadas. Esta función no recibe ningún parámetro y a partir del momento en el que se llama el comunicador ya no puede ser utilizado.

Función attach

La función *attach* enlaza una variable al comunicador. Una vez que una función está enlazada, no se debe usar excepto para desenlazarla o ejecutar un trabajo con *launchKernel*. Esta función tan solo recibe como parámetro un *Tile* (ver Capítulo 5) para enlazar (único tipo de variable que se puede enlazar en nuestro prototipo).

Función detach

La función *detach* desenlaza una variable del comunicador. Una vez que una función está desenlazada, se puede usar libremente. Esta función tan solo recibe como parámetro un *Tile* (ver Capítulo 5) que corresponda a una variable ya enlazada para desenlazar.

Función createInternal

La función *createInternal* crea una variable sin imagen fuera del comunicador llamadas "Variables Internas". Estas variables tan solo puede ser usada por el comunicador y los kernels que se lanzen a través de el. Las "Variables Internas", una vez creadas, se pueden usar como

si fueran variables enlazadas. Esta función tan solo recibe como parámetro un *Tile* (ver Capítulo 5) al que no se le haya asignado memoria pero si se le haya asignado un dominio.

Función destroyInternal

La función *destroyInternal* elimina los recursos asignados a una "Variable Interna". Esta función tan solo recibe como parámetro un *Tile* (ver Capítulo 5) correspondiente a una "Variable Interna".

Funcion launch

Esta función se encarga de añadir a la cola un kernel para ser ejecutado en cuanto sea posible, es decir, cuando llegue su turno en la cola de ejecución. Para ello, recibe como parámetros el kernel que tiene que ejecutar, los hilos con los que desea el programador ejecutar el kernel (típicamente el tamaño de los datos), el número de parámetros del kernel y los própios parámetros del kernel. La inclusion de un parámetro que represente el número de parámetros que se pasan se necesita debido a que esta función puede recibir un número indeterminado de parámetros para los kernels.

8.1.2. Estructura CommCPU

La estructura *CommCPU* es la estructura de comunicador especifica para controlar CPUs. En principio esta estructura controlaría un grupo de CPUs, sin embargo, en el prototipo se implementará para que un comunicador controle todas las CPU.

Para controlar las CPU y que permita ejecutar las diferentes tareas de forma asíncrona como si fuera un dispositivo, se ha pensado en utilizar una cola de tareas que se irán ejecutando independientemente del programa principal (hilo principal). Dicha cola de tareas, se almacena en esta estructura.

Las funciones que usan esta estructura, excepto las detalladas a continuación, son las mismas que *Comm* pero con una implementación específica para esta estructura.

Función addTask

La función *addTask* tiene como finalidad el añadir un kernel a la cola de tareas de CPU. Esta función recibe como parámetro el número de hilos, el kernel a ejecutar y la lista de parámetros del kernel.

Función addRef

La función *addRef* añade una referencia que indica que un tile está siendo usado, o se va a usar, para la ejecución de un kernel, por tanto, dicho tile no podrá ser desenlazado o eliminado, en caso de que sean variables enlazadas o variables internas respectivamente, hasta que la referencia se elimine. Esta función recibe como parámetro el tile al que se le tiene que añadir la referencia.

Función delRef

La función *delRef* elimina una referencia indicando que el tile ya a sido usado. Sin embargo, hasta que el número de referencias no llegue a cero no podrá ser desenlazado o eliminado. Esta función recibe como parámetro el tile al que se le tiene que eliminar la referencia.

Función worker

La función *worker* es una función que se ejecuta de forma concurrente al resto del programa y se encarga de ejecutar los kernels que hay en la cola. Esta función es un consumidor en el modelo de productor-consumidor donde obtiene kernels de la cola y los ejecuta.

8.1.3. Estructura CommGPU

La estructura *CommGPU* es la estructura de comunicador especifica para controlar GPUs. Un comunicador de GPU controla una única GPU.

Para administrar las transferencias de memoria, se crean imágenes de las variables enlazadas durante el enlazado y se realizan las transferencias. Dichas imágenes se tienen que localizar, para ello se almacena un puntero junto con la variable así como un puntero a los datos que contiene el tile para evitar transferencias innecesarias. La recuperación de los datos, se realiza durante la operación de desenlazado. Aunque el prototipo actual lo realiza de forma sincrona, el modelo contempla la posibilidad de que estas transferencias se realizaran de forma asíncrona.

Función getDevTile

La función *getDevTile* obtiene un puntero a la imagen en el dispositivo del tile enlazado en el host recibido como parámetro. Esta función recibe como parámetro un *Tile* (ver Capítulo 5) correspondiente a una variable Interna o a una variable enlazada y devuelve un puntero a la variable del dispositivo.

8.2. Modelo dinámico

En esta sección se se va a mostrar como realizan su labor las funciones del prototipo junto con una breve descripción del proceso. Para ello se utilizará un diagrama de secuencia que corresponda con el caso de uso descrito en la sección 7.2. Dicho diagrama de secuencia, descrito en la Figura 8.2, servirá como base para todos los diagramas de secuencia y para entender el funcionamiento de los comunicadores.

8.2.1. Creación y Destrucción de Comunicadores

En la Figura 8.3 se muestra como se crea un comunicador. La función de creación recibe el tipo de dispositivo que va a utilizar este comunicador junto con el numero de dispositivo. A continuación llama a la función de creación concreta del tipo de dispositivo seleccionado.

La creación de un comunicador de CPU implica que se debe iniciar la ejecución de un hilo encargado de ejecutar aquellos kernels que se quiera que se ejecuten en la CPU. Este hilo, cuyo flujo de trabajo se puede ver en la Figura 8.4, lo único que hace es ejecutarse mientras

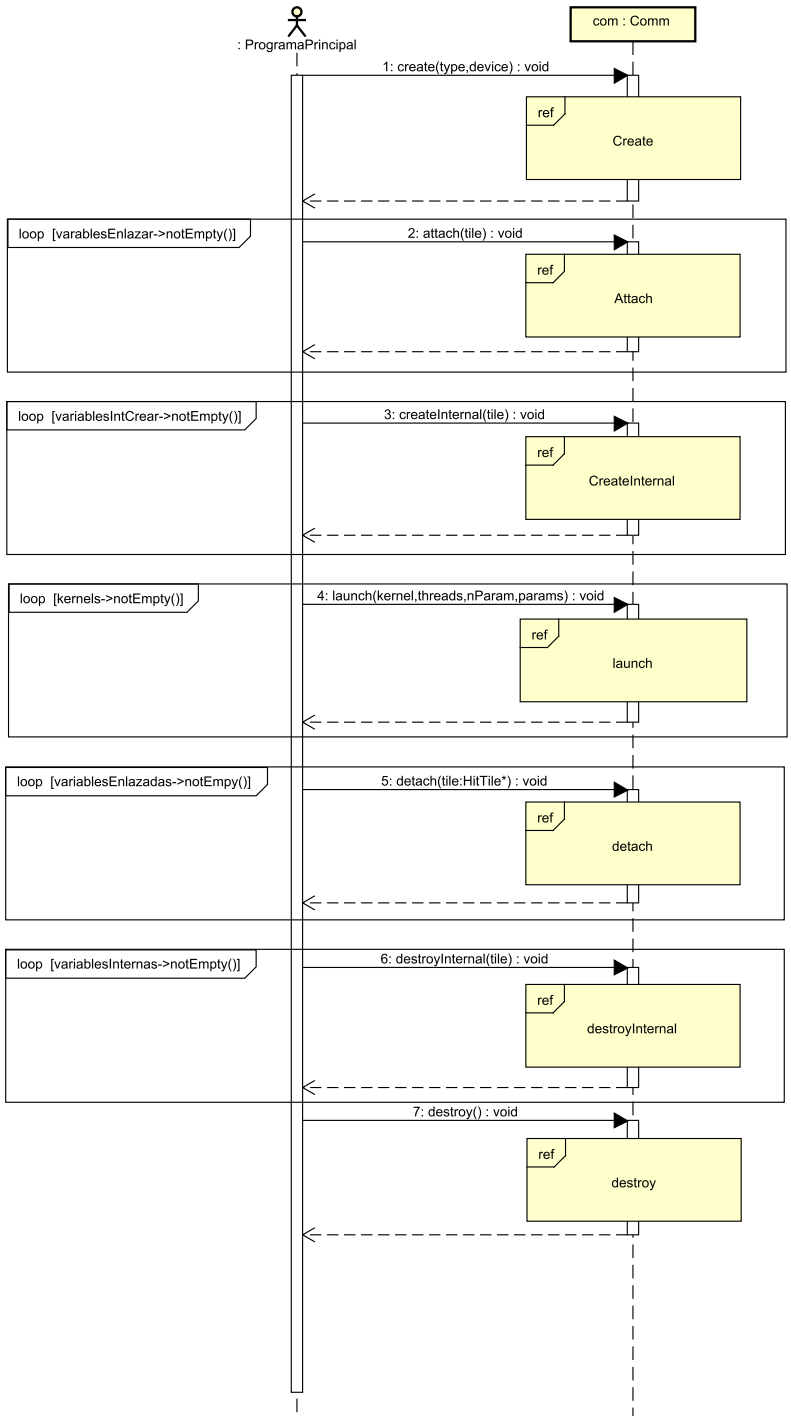

Figura 8.2: Diagrama de secuencia de la creación de un comunicador

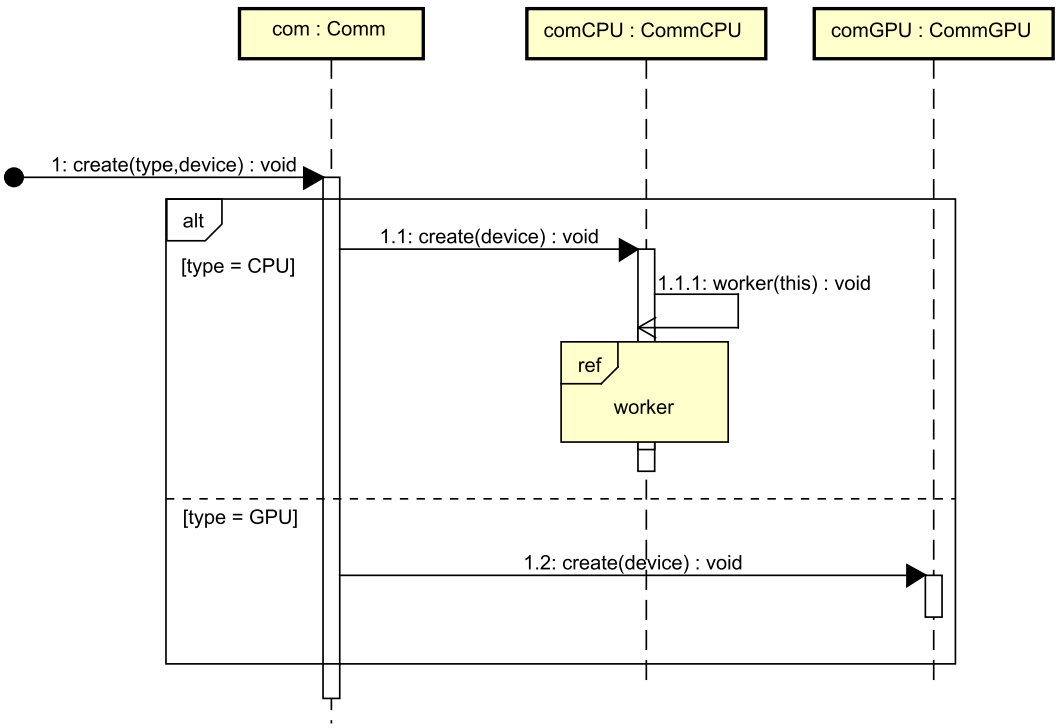

Figura 8.3: Diagrama de secuencia de la creación de un comunicador

no se le diga que se pare. Además, cuando se le ordena que pare, este continuará ejecutándose mientras le queden tareas pendientes.

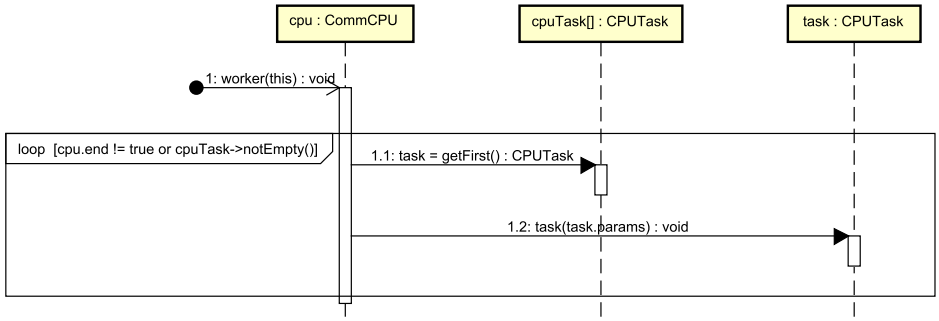

Figura 8.4: Diagrama de secuencia del hilo worker en un comunicador de CPU

La función de destrucción dependiendo del valor de la variable type, llama a la función de destrucción correspondiente. Esto se puede ver en la Figura 8.5

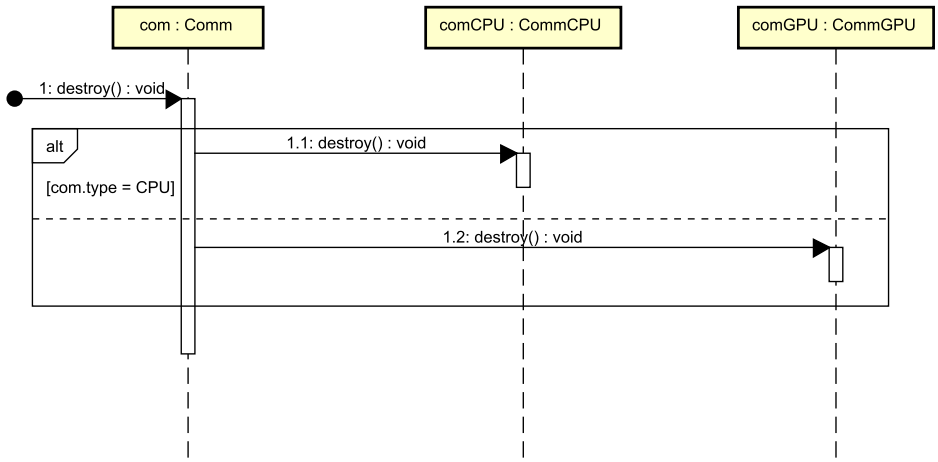

Figura 8.5: Diagrama de secuencia de la destrucción de un comunicador

8.2.2. Enlazado y Desenlazado de variables

El enlazado de variables se puede ver en la Fig. 8.6. La secuencia de enlazado depende del tipo de dispositvo, donde en la CPU sólo se guarda en una lista de variables enlazadas; mientras que en GPU se crea la imagen en el dispositivo, realizando transferencia de los datos, y se almacenan tanto la imagen como la variable.

En la Fig. 8.7 se puede ver el funcionamiento del desenlazado de variables. En el desenlazado de una variable enlazada a un comunicador de CPU simplemente es esperar a que la variable ya no se utilice (mediante mecanismos de sincronizacion entre hilos) y sacarlo de

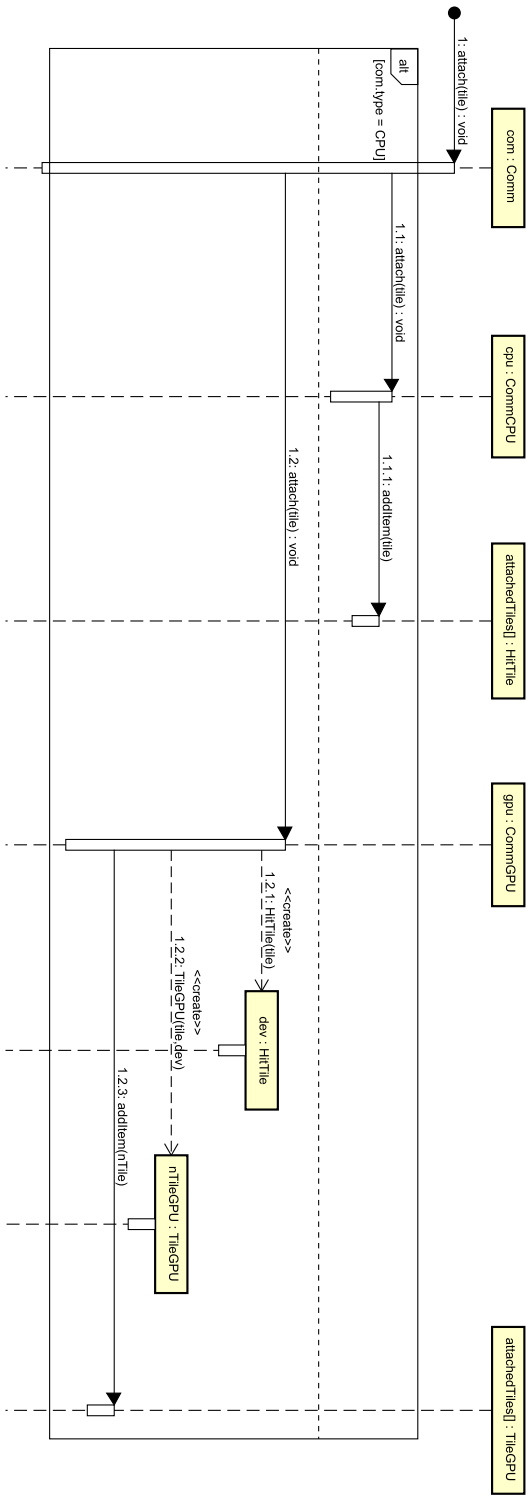

Figura 8.6: Diagrama de secuencia de enlazado de una variable

la lista de variables enlazadas. Si la variable está enlazada a la GPU, se debe encontrar su correspondencia en la lista de tiles enlazados y eliminar su imagen, realizando primero las transferencias necesarias.

8.2.3. Creación y destrucción de variables internas

La creación de variables internas, como se puede ver en la Figura 8.8 su acción depende de si el comunicador es de CPU o GPU. Si el comunicador es de CPU simplemente le manda reservar la memoria que necesita y se almacena. Sin embargo, si el comunicador es de GPU se crea un tile en la GPU y se le reserva la memoria, y el tile del dispositivo y el del host se almacenan.

Para la destruccion de las variables internas, en la Figura 8.9 se puede ver que para CPU simplemente se busca el tile, se le saca de la lista y se libera la memoria. Para el tile de GPU, al igual que el de CPU se busca el tile en la lista, se le saca y se le libera la memoria; solo que en este caso, la memoria que se libera es la del dispositivo.

8.2.4. Ejecución de kernels

La ejecución de kernels es una acción un tanto compleja puesto que la forma en la que actúa depende de una serie de parámetros: el tipo de comunicador y el rol de las variables. Para su realización la función de lanzamiendo de los kernels recibe el propio kernel que tiene que ejecutar, los hilos con los que tiene que ejecutarlos, el número de parámetros que se le envían y los propios parámetros. Esta función selecciona el método más apropiado dependiendo de su tipo.

En el caso de un comunicador de CPU lo primero que se hace es añadir una referencia a todos los parámetros recibidos. Esto impedirá que las variables referenciadas se puedan eliminar o desenlazar mientras se están utilizando. Seguidamente se añadirá una nueva tarea a la lista de tareas para su ejecución. Esta lista de tareas usa un algoritmo de planificación FCFS (First Come, First Service).

En el caso de un comunicador de GPU lo primero que se hace es transferir al dispositivo aquellos tiles correspondientes al rol del parámetro IN o IO y que aún no hayan sido transferidos. Posteriormente se ordenará al acelerador que ejecute el kernel. El acelerador no lo ejecutará directamente si no que lo pondrá en una lista de ejecución la cual funciona utilizando una planificación FCFS.

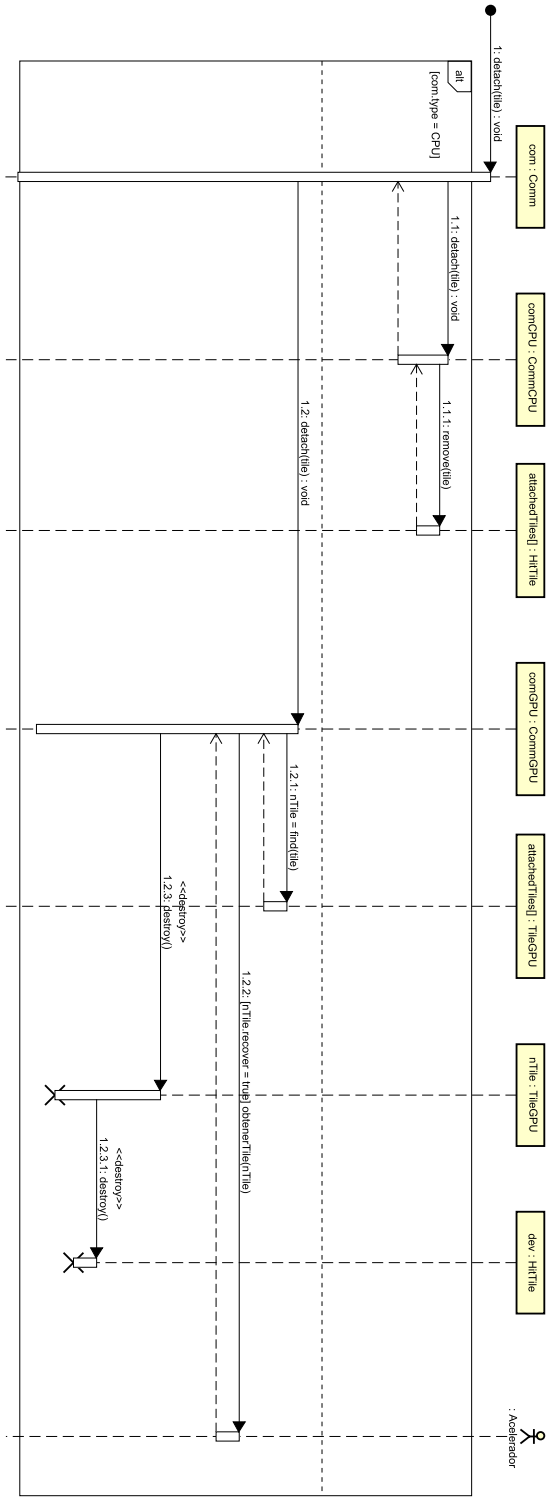

Figura 8.7: Diagrama de secuencia de desenlazado de una variable

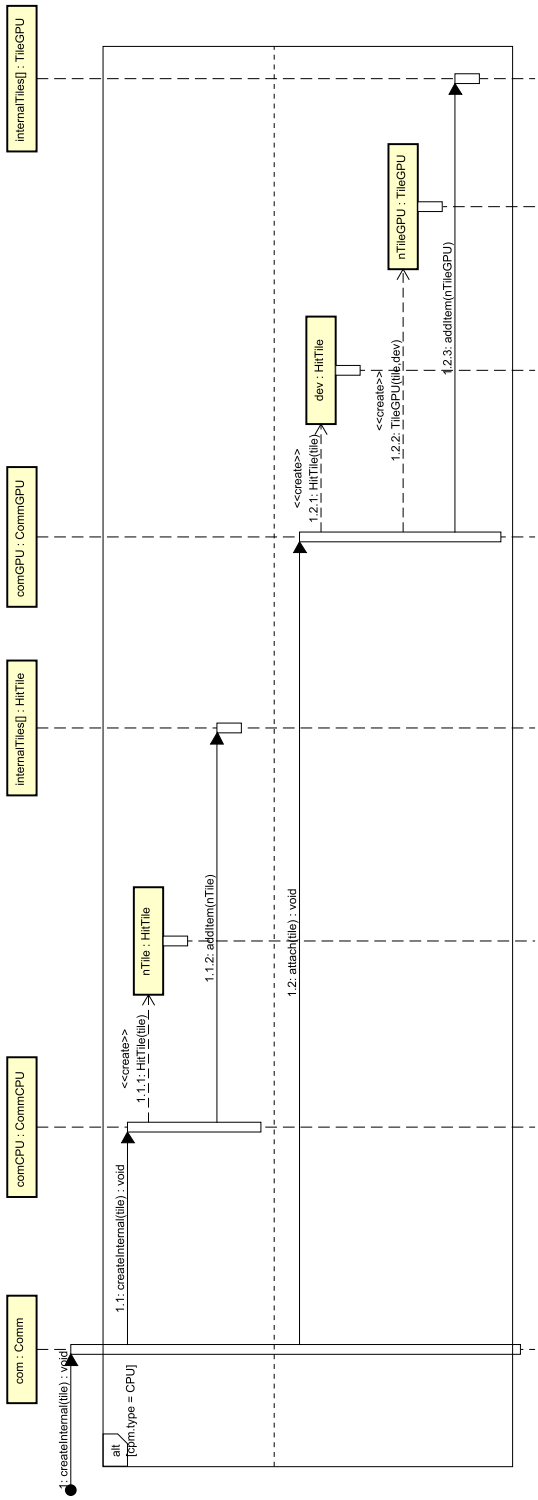

Figura 8.8: Diagrama de secuencia de creación de variables internas

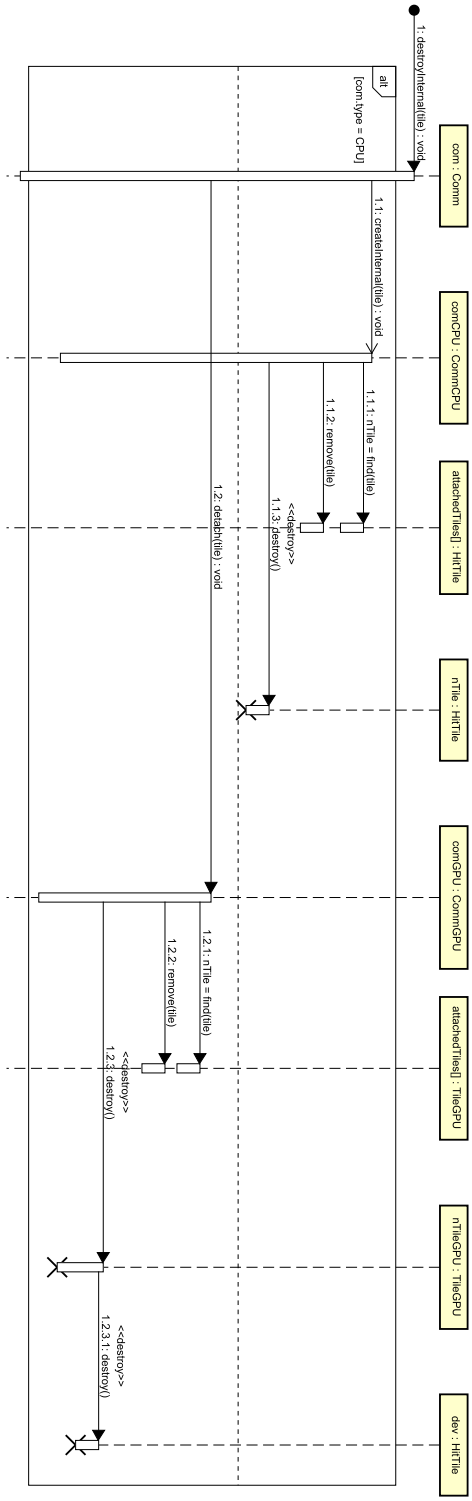

Figura 8.9: Diagrama de secuencia de destrucción de variables internas

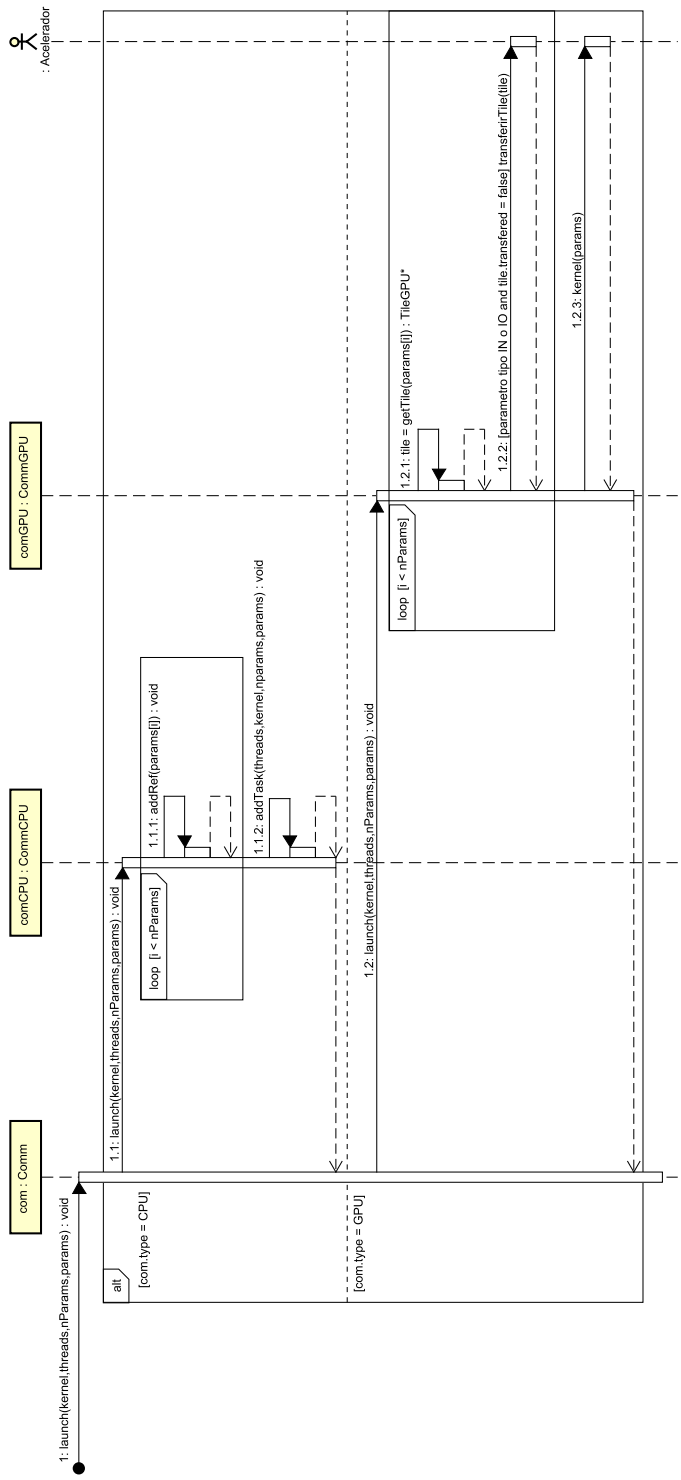

Figura 8.10: Diagrama de secuencia del lanzamiento de un kernel

Parte IV

Implementación y Pruebas

Capítulo 9

Implementación del modelo

Durante este capítulo se explicarán las diferentes decisiones tomadas para la implementación del proyecto como las herramientas, porqué se ha elegido el uso de macros en algunas partes de la implementación o qué estructuras de datos se han utilizado. Además, también se explicará como se han implementado algunas de las partes más importantes del prototipo.

9.1. Herramienta de generación de código

Con motivo de automatizar la compilación se usará una herramienta reconocida llamada make. Make es una herramienta que nos permite mediante el uso de un único comando la compilación de un proyecto complejo, para ello usa un archivo llamado Makefile como plantilla para generar el código ejecutable.

El uso de la herramienta make para proyectos de envergadura considerable tiene un gran problema debido a la complejidad del archivo Makefile. Por el motivo mencionado, se suelen utilizar herramientas de generación de proyectos como: autotools, cmake o scons; los cuales se han considerado para usar en este proyecto.

- Autotools: Es un conjunto de aplicaciones que facilitan la creación de de código fuente portable a varios sistemas Unix. Esta herramienta es autocontenida puesto que sólo necesita las herramientas estándares para la compilación del código. Por desgracia, es bastante complicada de aprender a usar.
- CMake: Es una serie de herramienta multiplataforma que permite la generación, pruebas y empaquetado de software. CMake utiliza ficheros de configuración muy sencillos, por lo que es facil de aprender, e independientes de plataforma. La única pega es que requiere disponer de su herramienta para la compilación del código fuente, aunque está disponible para la mayor parte de las distribuciones Unix y para Windows.
- SCons: es una herramienta de código abierto para la construcción e instalación de software a través de scripts hechos en Python para los sistemas operativos basados en Unix. Esta herramienta falla en que es muy lenta y requiere de una instalación completa de Python para funcionar.

Después de considerar estas tres opciones se ha decidido utilizar la herramienta CMake debido a sus numerosas ventajas. Especialmente, se ha considerado su simplicidad a la hora

de realizar tareas comunes, puesto que no se desea realizar en nuestro proyecto nada más que tareas simples pero en gran cantidad, que es totalmente multiplataforma y que está disponible en todas las distribuciones linux con dependencias mínimas.

9.2. Nombres de función y parámetros comunes

Para todas las funciones que pertenecen a una clase en el diseño (ver capítulo 8) se le ha añadido un prefijo con el nombre de la clase. Este prefijo es usado para distinguir unas funciones de otras, puesto que en el lenguaje C no se puede encapsular las funciones, y estas tienen que tener un nombre único. De esta forma si se desea llamar a la función de enlazado de un comunicador de CPU, el nombre de la función será: *CommCPUAttach*.

Por el mismo motivo que se le ha añadido el prefijo, C no es orientado a objetos, a las funciones se les tiene que pasar un parámetro extra, no contemplado en el diseño, el puntero a la estructura de datos. Este puntero, debe apuntar a una estructura de datos válida y del mismo tipo que la estructura con la que trabaja. Para inicializar estas variables existe la función *create* para cada estructura de tipo comunicador, la cual es la única a la que no se le tiene que pasar un puntero a una estructura inicializada, y actúa como el constructor de una clase. En la Figura 9.1 se pueden ver un ejemplo de las cabeceras de las funciones que utilizan a la estructura Comm.

```
1 /* Algunas de las funciones que usan la estructura Comm. */
2 CommCreate(Comm* comm, int device);
3 CommDestroy(Comm* comm);
4 CommAttach(Comm* comm, HitTile* tile);
5 CommDetach(Comm* comm, HitTile* tile);
```

Figura 9.1: Ejemplo de algunas de las cabeceras de funciones del prototipo.

9.3. Configuración de los kernels de GPU

Aquí se hablará de la forma en la que se ha implementado la configuración de los kernels, tanto para la configuración de la geometría de hilos, como para la asignación de los roles que posteriormente se utilizarán a la hora de gestionar las transferencias de datos. Dicha configuración se obtendrá a partir de los datos suministrados por el usuario a través de unas macros especialmente creadas con este propósito. Los datos suministrados por los usuarios serán la caracterización de los kernels y el rol de los parámetros.

9.3.1. Carácterización de los kernels

Para la implementación de la caracterización se ha definido la macro *KERNEL_CHAR*, basada en el modelo de parametrización de código de la herramienta TuCCompi [13], que se tiene que usar antes de la definición del kernel. Se ha implementado como una macro debido a que al disponer en el momento de compilación los datos necesarios para la configuración den función de los parámetros de entrada ya se pueden prefijar valores para una buena

configuración en tiempo de compilación, ahorrándonos el tener que calcularlo durante la ejecución. Esta macro recibe los siguientes parámetros:

1. Nombre del kernel.

2. Número de dimensiones del conjunto de hilos con los que se trabaja: Los valores pueden ser 1, 2 o 3. Estos valores corresponden con las dimensiones que puede tener un kernel desarrollado para una GPU.

3. Patrón de acceso a memoria: Indica la forma en la que los hilos de la GPU accederán a la memoria. Se definen tres posibles valores:

   - Full: Indica que los accesos a memoria serán totalmente coalescentes.
   - Medium: Indica que hay partes en las que los accesos a memoria no son totalmente coalescentes.
   - Scatter: Indica que los accesos a memoria no tienen ninguna coalescencia.

4. Ratio de carga computacional: Es el ratio entre el numero de operaciones aritméticas y/o lógicas con respecto al número de accesos a memoria global. Se definen tres valores:

   - Low: El ratio esta entre 0 y 10 operaciones por acceso a memoria.
   - Medium: El ratio esta entre 10 y 100 operaciones por acceso a memoria.
   - High: El ratio esta por encima de 100 operaciones por acceso a memoria.

5. Ratio de reutilización de datos entre hilos: Es un ratio que compara la reutilización de los datos entre hilos y los accesos a memoria. Se definen tres valores:

   - Low: Cuando no hay reutilización de datos entre los hilos.
   - Medium: Todas aquellas situaciones entre Low y High.
   - High: Cuando todos los hilos cogen todos los valores usados por todos los hilos.

```
1 KERNEL_CHAR(kernelA,1,def,def,def)
2 KERNEL_CHAR(kernelB,1,full,low,high)
```

Figura 9.2: Ejemplo de caracterización de kernels

En la Figura 9.2 se puede ver el uso de las macros para caracterizar dos kernels que utilizan una única dimension de hilos. En el kernel A (línea 1) se le ha dado una configuración por defecto. Mientras que al kernel B (línea 2) se le ha dado una configuración que indica que su código el totalmente coalescente, el ratio entre operaciones aritméticas y de memoria es menor que 10 y que hay un gran reutilización en los datos.

Los diferentes parámetros, los rangos utilizados y los valores óptimos de configuración devueltos han sido tomados de los modelos presentados a la comunidad científica en [23, 15, 14].

9.3.2. Asignación del rol de los parámetros

Para la asignación del rol de los parámetros se ha creado la macro *KERNEL_GPU* que sustituye a la cabecera de la función. Esta pseudocabecera permite entre otras cosas indicar el rol de los parámetros junto al propio parámetro, funcionalidad que existe en otros lenguajes, lo cual es mucho más visual para el programador a la hora de usarlo. Esta macro a su vez se encarga de declarar unos indices de hilo comunes para GPU y CPU. El macro oculta la sintaxis propia para los kernels de CUDA. Cuando se utiliza esta macro es obligatorio utilizar punteros y tiles de Hitmap.

```
1 KERNEL_GPU(kernel, OUT, HitTile_float*, dst, \
2                         IN, HitTile_float*, src){
3         /* Kernel code. */
4 }
```

Figura 9.3: Ejemplo de cabecera de un kernel GPU.

En la Figura 9.3 se puede ver el uso de la pseudocabecera para un kernel de GPU. Este kernel tiene como parámetro dos tiles: el primero, es de salida (OUT) y es de tipo float; y el segundo, es de entrada (IN) y de tipo float.

9.4. Unificación de los paradigmas CPU-GPU

La unificación de los paradigmas de CPU y GPU es probablemente una de las mayores distinciones de la programación con Comunicadores, puesto que nos permite aplicar el mismo algoritmo que el utilizado para GPU en la CPU con unos cambios mínimos. En este proyecto se ha elegido tomar una perspectiva contraria a la dada por OpenACC [3], la cual transforma el código de CPU para adaptarlo a los aceleradores, y utilizar el modelo de programación de GPU para el código de CPU. Para ello, un comunicador de CPU tiene que simular una serie de mecanismos propios del manejo de GPU: el lanzamiento asíncrono de los kernels y la ejecución del kernel en una cantidad masiva de hilos paralelos.

9.4.1. Lanzamiento asíncrono de kernels

Para el lanzamiento asíncrono de los kernels se utiliza un hilo, llamado en el diseño worker (ver capítulo 8), que actúa como un consumidor de tareas de una cola. Por tanto aquí se está aplicando un modelo productor-consumidor donde el productor es el programa principal, que añade tareas a la cola durante el lanzamiento del kernel, y el consumidor es el hilo del comunicador, el cual se encarga de ejecutarlas. Para implementar esto, tanto el hilo como las sincronizaciones necesarias en el modelo de productor-consumidor, se han utilizado las funciones de la biblioteca *pthreads*.

Antes de añadir un kernel a la cola es necesario añadir las referencias a sus parámetros. Estas referencias se utilizan como mecanismos de sincronización que impiden que dichas variables se desenlace o destruya, dependiendo de si es una variable enlazada o una variable interna, antes de haber terminado su ejecución. Dichas referencias se eliminan tras su ejecución permitiendo, en caso de que el numero de referencias llegue a 0, eliminar o desenlazar dicha variable. Para bloquear el programa principal durante el desenlazado/destrucción de

una variable se utilizan los cerrojos condicionales de pthreads puesto que son más eficientes que el uso de semáforos.

Para la ejecución de un kernel el hilo debe antes seleccionar cuál debe ejecutar, de entre los que hay en la cola de tareas, para ello emplea un algoritmo de planificación. Para este prototipo se ha elegido un algoritmo de planificación FCFS (First Come, First Service) debido a su sencillez y que para el nivel de complejidad de las pruebas a las que se someterá, este es suficientemente eficaz. La cola de tareas se ha implementado como una lista enlazada donde se almacena un puntero a la función que ejecuta el kernel y su lista de parámetros. Esto nos permite añadir un número ilimitado de tareas con un coste de inserción y de extracción de $O(1)$. Para la selección de la estructura de datos se ha tenido en cuenta el algoritmo de planificación, por lo que si este se cambiara habría que considerar otro tipo de estructuras.

9.4.2. Emulación de hilos de GPU

La implementación de los hilos de GPU en la CPU para la ejecución en paralelo de los kernels se ha conseguido mediante el uso de una serie de bucles anidados, los cuales representan las dimensiones en un conjunto de hilos de GPU, donde se ejecuta el kernel proporcionado n-veces ajustando previamente el identificador de hilo que se le pasa al kernel. Para la paralelización de estos bucles se utiliza OpenMP [4] el cual divide cada una de las vueltas de los bucles en secciones de código que se distribuyen entre una serie de hilos para una ejecución en paralelo. Esto permite que se ejecuten n-kernels en k-hilos proporcionando aún más la sensación de que son kernels que se ejecutan en una GPU.

```
1 #pragma omp parallel for collapse(3) //Directiva OpenMP
2 for (int i = 0; i < threads.z; i++){
3         for (int j = 0; j < threads.y; j++){
4                 for (int k = 0; k < threads.x; k++){
5                         kernel(i,j,k,params...);
6                 }
7         }
8 }
```

Figura 9.4: Posible código para la paralelización de los kernels.

En la Figura 9.4 se puede ver un posible código para la paralelización de un kernel en un conjunto de hilos de tres dimensiones. Se utiliza una directiva de OpenMP para la paralelización. Un código similar a este, se encuentra implementado en la macro *KERNEL_CPU* (ver Figura 9.5) la cual sustituye a la cabecera de la función. Dicha macro tiene una forma similar a su versión de GPU, con motivo de que sea más fácil de usar para el usuario, pero con funcionalidades totalmente diferentes puesto que esta macro lo que hace es crear dos funciones ad-hoc: la función del kernel y una función de envoltura.

- Función del kernel: es la función definida por el usuario al que se le añade un parámetro extra. Este parámetro es el identificador del hilo para su uso dentro del kernel.

- Función de envoltura: esta función es la que realmente ejecutará el worker (ver sección 9.4.1) y es la que contiene los bucles que emulan los hilos de GPU, los cuáles ejecutan la función del kernel. El motivo para crear esta función es para evitar que los

bucles paralelos ejecuten un puntero a función, puesto que este tipo de ejecuciones conllevan a un overhead mayor, ya que el compilador no le es posible optimizar la llamada a la función.

```
1 KERNEL_CPU(kernel, OUT, HitTile_float*, dst, \
2                          IN, HitTile_float*, src){
3         /* Kernel code. */
4 }
```

Figura 9.5: Ejemplo de cabecera de un kernel CPU.

9.5. Mapeo de variables

Para poder utilizar las mismas variables para el uso normal dentro del programa principal tanto como para el lanzamiento de los kernels, es necesario crear una correspondencia entre la variable y su imagen en el dispositivo. En el diseño se creó una estructura que contiene estos datos y se almacena en unas listas dentro del comunicador (ver capítulo 8). Dichas listas están implementadas como arrays de un tamaño fijo bastante grande junto con una variable entera para saber el número de elementos en el array. El motivo para utilizar un array es debido a que las búsquedas son más rápidas en este tipo de estructuras que las listas enlazadas y además no se espera que haya un número excesivamente grande de variables. A esto hay que añadir que su implementación es sencilla.

Para las búsquedas dentro de dichas listas a partir del tile original, o un puntero a este, se utiliza la dirección de memoria que se compara con la dirección de memoria almacenada previamente, es decir, se usa la dirección de memoria como campo clave. Estas búsquedas se realizan para la obtención de la imagen de la variable en el dispositivo (GPUs), modificaciones en el número de referencias (CPUs) y enlazado/creación y desenlazado/destrucción de las varables.

En el caso de las variables enlazadas y las variables internas en la CPU cuando se intenta desenlazar/destruir una de estas variables, se deben controlar las referencias, es decir el numero de veces que se utiliza dicha variable, y esperar a que el número de referencias sea 0. Para lograr esto se utilizan los cerrojos condicionales de pthreads que detienen el programa principal hasta que se cumple la condición dada.

9.6. Lanzamiento de kernels

Para el lanzamiento de los kernels se utiliza la macro *KernelLaunch*. Esto se debe a que para realizar el lanzamiento de los kernels se debe realizar una serie de operaciones en tiempo de compilación. La principal de estas operaciones es selección de la geometría del bloque, la cual se obtiene a partir de la caracterización. Esta operación solo se puede realizar en tiempo de compilación del programa que utiliza el prototipo puesto que antes no se dispone de la información necesaria. Otra de las operaciones es la obtención de los roles de las variables de un kernel, puesto que el nombre de las variables donde se almacena, son dependientes del nombre del kernel.

Además se han definido las macros de tal forma que se ha procurado desvincular el funcionamiento de los comunicadores del SDK de CUDA cuando no se definen kernels de GPU. Por tanto, en caso de no haya kernels GPUs se puede utilizar otro compilador en vez del compilador de CUDA.

Capítulo 10

Pruebas sobre el prototipo

Las pruebas son conjuntos de actividades que pueden planearse por adelantado y realizarse de manera sistemática. En este capítulo se realizará una serie de pruebas sobre el prototipo con el objetivo de validar que no hay fallos en la implementación del mismo.

En este capítulo primero se realizará un plan de pruebas donde se realizará una descripción detallada de cada uno de los casos de pruebas que se aplicarán para la validación del prototipo. Posteriormente, se mostrarán los resultados de las pruebas obtenidos para cada una de las versiones por las que ha pasado el prototipo.

10.1. Plan de pruebas

En esta sección se identificarán las distintas pruebas que se efectuarán al prototipo de forma que se pueda validar la correcta implementación del mismo. Estas pruebas se aplicarán a los distintos prototipos creados. Cada versión del prototipo tendrá el formato $x.y$ donde x corresponderá a la iteración en la que fue creado, e y corresponde a la revisión del prototipo, es decir, las veces que se ha modificado en esa iteración antes de esa versión.

Estas pruebas se dividen en: Pruebas unitarias, Pruebas de integración y Pruebas de validación. En las pruebas se indicará qué se prueba, la entrada y la salida esperada. Además, también se indicará la versión del prototipo a partir de la que es aplicable la prueba.

10.1.1. Pruebas de unidad

Las pruebas de unidad [16] enfocan los esfuerzos de verificación en la unidad más pequeña del diseño de software: el componente o módulo de software. Al usar la descripción del diseño de componente como guía, las rutas de control importantes se prueban para descubrir errores dentro de la frontera del módulo. Cuando hay módulos subordinados, se utilizan representantes que lo único que realizan es dar salidas con una manipulación mínima de los datos, permitiendo la ejecución sin los módulos reales. La relativa complejidad de las pruebas y los errores que descubren están limitados por el ámbito restringido que se establece para la prueba de unidad. Las pruebas de unidad se enfocan en la lógica de procesamiento interno y de las estructuras de datos dentro de las fronteras de un componente. Este tipo de pruebas puede realizarse en paralelo para múltiples componentes.

En el caso de nuestro prototipo se realizan para todas las funciones una serie de pruebas donde a partir de una entrada concreta se debe obtener una salida esperada. Estas pruebas se han realizado por cada una de las funciones descritas en el diseño (ver capítulo 8) que tengan entrada y salida de datos. Las pruebas se pueden ver en los Cuadros 10.1 y 10.2.

10.1.2. Pruebas de integración

Las pruebas de integración [16] son una técnica sistemática para construir la arquitectura del software mientras se llevan a cabo pruebas para descubrir errores asociados con la interfaz. El objetivo es tomar los componentes probados de manera individual y construir una estructura de programa que se haya dictado por diseño. Esta integración puede ser descendiente o ascendente, es decir, se van integrando primero los módulos de subordinados más altos o mas bajos en la jerarquía. Esta prueba sólo se ejecuta una vez que las pruebas de unidad resultan exitosas.

Para el caso de nuestro prototipo se realizarán primero las mismas pruebas que las descritas para las pruebas de unidad, sin embargo, internamente se utilizarán la funciones auténticas, comprobando así que todo funciona correctamente. La integración será ascendente. Además se añadirán pruebas para los casos no contemplados en las pruebas de unidad. Las pruebas añadidas se pueden ver en el Cuadro 10.3.

10.1.3. Pruebas de validación

Las pruebas de validación [16] comienzan en la culminación de las pruebas de integración, cuando se probaron componentes individuales, el software está completamente ensamblado como un paquete y los errores de interfaz se descubrieron y corrigieron. Las pruebas se enfocan a las acciones visibles para el usuario y las salidas del sistema reconocibles por el usuario.

En el caso de nuestro prototipo, las pruebas de validación consisten en utilizar la biblioteca en una serie de aplicaciones representativas y asegurarse de que los cálculos realizados son correctos. Para nuestras pruebas se utilizaran las mismas aplicaciones que para los casos de estudio en la sección 11.3 a las que se les introducirá una entrada concreta y deben producir el resultado correcto. Un resumen de estas pruebas se puede ver en el Cuadro 10.4. Hay que tener en cuenta que en la primera iteración el prototipo no fue funcional por lo que las pruebas de validación no proceden y que los experimentos de la suma de matrices y la multiplicación de matrices, no se añadieron hasta la tercera iteración, aunque sean compatibles con la versión de la segunda iteración.

10.2. Resultados de las pruebas

A continuación se van a mostrar los resultados de todas las pruebas de las diferentes versiones del prototipo. Los posibles resultados de estas pruebas pueden ser OK, que significa que pasa la prueba, y FAIL, que significa que no la pasa. Cuando no se pasa una prueba, se buscan los problemas que hacen que se produzcan todos los errores y se solucionan. Una vez solucionados los problemas se vuelven a realizar las pruebas. En el caso que el resultado se muestre en blanco, indica que esta prueba no procede en esa versión del prototipo. Los resultados de las pruebas de unidad se pueden ver en el Cuadro 10.5, los de las pruebas de integración en el Cuadro 10.6 y los de las pruebas de validción en el Cuadro 10.7.

Cod.	Función	Entrada	Salida esperada	Ver.
PU-1	CommCreate	Una estructura Comm no inicializada, el tipo de dispositivo y un número de dispositivo.	El valor de device y del tipo de dispositivo de la estructura tienen que ser igual al los de la entrada.	1.0
PU-2	CommGPUCreate	Una estructura CommGPU no inicializada.	Las listas de tiles tienen que tener la memoria reservada y estar vacías.	2.0
PU-3	CommGPUDestroy	Una estructura CommGPU inicializada.	Las listas de tiles no tienen que tener memoria reservada.	2.0
PU-4	CommGPUAttach	Una estructura CommGPU no vacía y un tile ni enlazado ni interno.	El tile tiene que estar en la lista de tiles enlazados en la última posición.	2.0
PU-5	CommGPUDetach	Una estructura CommGPU no vacía y un tile enlazado.	El tile no tiene que estar en la lista de tiles enlazados.	2.0
PU-6	CommGPUCreateInternal	Una estructura CommGPU no vacía y un tile ni enlazado ni interno.	El tile tiene que estar en la lista de tiles internos en la última posición.	2.0
PU-7	CommGPUDestroyInternal	Una estructura CommGPU no vacía y un tile interno.	El tile no tiene que estar en la lista de tiles internos.	2.0
PU-8	getTile (CommGPU)	Una estructura CommGPU no vacía y un tile ni enlazado ni interno.	Devuelve NULL.	2.0
PU-9	getTile (CommGPU)	Una estructura CommGPU no vacía y un tile enlazado.	Devuelve un TileGPU correspondiente.	2.0
PU-10	getTile (CommGPU)	Una estructura CommGPU no vacía y un tile interno.	Devuelve un TileGPU correspondiente.	2.0
PU-11	CommCPUCreate	Una estructura CommCPU no inicializada.	Las listas de tiles tienen que tener la memoria reservada y estar vacías, la lista de tareas tiene que estar vacía, la variable end tiene que estar a false.	3.0

Cuadro 10.1: Pruebas de unidad

Cod.	Función	Entrada	Salida esperada	Ver.
PU-12	CommCPUDestroy	Una estructura CommCPU inicializada.	Las listas de tiles no tienen que tener memoria reservada, la lista de tareas tiene que estar vacía y la variable end tiene que estar a true.	3.0
PU-13	CommCPUAttach	Una estructura CommCPU no vacía y un tile no enlazado.	El tile tiene que estar en la lista de tiles enlazados en la última posición con el número de referencias a 0.	3.0
PU-14	CommCPUDetach	Una estructura CommCPU no vacía y un tile enlazado.	El tile no tiene que estar en la lista de tiles enlazados.	3.0
PU-15	CommCPUCreateInternal	Una estructura CommCPU no vacía y un tile.	El tile tiene que estar en la lista de tiles internos en la última posición.	3.0
PU-16	CommCPUDestroyInternal	Una estructura CommCPU no vacía y un tile enlazado.	El tile no tiene que estar en la lista de tiles internos con el número de referencias a 0.	3.0
PU-17	addTask (CommCPU)	Una estructura CommCPU no vacía, un shape con la geomotría de hilos, puntero a una función y una lista de parámetros.	La lista tiene una nueva tarea que corresponde con los datos de entrada.	3.0
PU-18	addRef (CommCPU)	Una estructura CommCPU no vacía y un tile enlazado.	La referencia del tile en la lista de enlazados se incrementa en 1.	3.0
PU-19	addRef (CommCPU)	Una estructura CommCPU no vacía y un tile interno.	La referencia del tile en la lista de internos se incrementa en 1.	3.0
PU-20	delRef (CommCPU)	Una estructura CommCPU no vacía y un tile enlazado.	La referencia del tile en la lista de enlazados se decrementa en 1.	3.0
PU-21	delRef (CommCPU)	Una estructura CommCPU no vacía y un tile interno.	La referencia del tile en la lista de internos se decrementa en 1.	3.0

Cuadro 10.2: Pruebas de unidad (cont.)

Cod.	Función	Entrada	Salida esperada	Ver.
PI-22	CommDestroy	Una estructura Comm inicializada.	Ninguno de los comunicadores debe estar inicializado.	2.0
PI-23	CommAttach	Una estructura Comm de GPU no vacía y un tile ni enlazado ni interno.	El tile tiene que estar en la lista de tiles enlazados, en la estructura correspondiente al del tipo GPU, en la última posición.	2.0
PI-24	CommAttach	Una estructura Comm de CPU no vacía y un tile ni enlazado ni interno.	El tile tiene que estar en la lista de tiles enlazados, en la estructura correspondiente al del tipo CPU, en la última posición.	3.0
PI-25	CommDetach	Una estructura Comm de GPU no vacía y un tile enlazado.	El tile no tiene que estar en ninguna la listas de tiles enlazados.	2.0
PI-26	CommDetach	Una estructura Comm de CPU no vacía y un tile enlazado.	El tile no tiene que estar en ninguna la listas de tiles enlazados.	3.0
PI-27	CommCreateInternal	Una estructura Comm de GPU no vacía y un tile ni enlazado ni interno.	El tile tiene que estar en la lista de tiles internos, en la estructura correspondiente al del tipo GPU, en la última posición.	2.0
PI-28	CommCreateInternal	Una estructura Comm de CPU no vacía y un tile ni enlazado ni interno.	El tile tiene que estar en la lista de tiles internos, en la estructura correspondiente al del tipo CPU, en la última posición.	3.0
PI-29	CommDestroyInternal	Una estructura Comm de GPU no vacía y un tile interno.	El tile no tiene que estar en ninguna de las listas de tiles internos.	2.0
PI-30	CommDestroyInternal	Una estructura Comm de CPU no vacía y un tile interno.	El tile no tiene que estar en ninguna de las listas de tiles internos.	3.0

Cuadro 10.3: Pruebas de integración

Cod.	Prueba	Versión
PV-1	PDE Jacobi: GPU	2.0
PV-2	PDE Jacobi: CPU	3.0
PV-3	Suma de matrices: GPU	2.0
PV-4	Suma de matrices: CPU	3.0
PV-5	Multiplicación de matrices: GPU	2.0
PV-6	Multiplicación de matrices: CPU	3.0

Cuadro 10.4: Pruebas de validación

Cod.	v1.0	v2.0	v2.1	v3.0	v3.1	v3.2	v3.3	v3.4
PU-1	OK	OK	OK	OK	OK	OK	OK	OK
PU-2		OK	OK	OK	OK	OK	OK	OK
PU-3		OK	OK	OK	OK	OK	OK	OK
PU-4		OK	OK	OK	OK	OK	OK	OK
PU-5		FAIL	OK	OK	OK	OK	OK	OK
PU-6		OK	OK	OK	OK	OK	OK	OK
PU-7		FAIL	OK	OK	OK	OK	OK	OK
PU-8		OK	OK	OK	OK	OK	OK	OK
PU-9		OK	OK	OK	OK	OK	OK	OK
PU-10		FAIL	OK	OK	OK	OK	OK	OK
PU-11				FAIL	OK	OK	OK	OK
PU-12				OK	OK	OK	OK	OK
PU-13				OK	OK	OK	OK	OK
PU-14				OK	OK	OK	OK	OK
PU-15				OK	OK	OK	OK	OK
PU-16				OK	OK	OK	OK	OK
PU-17				FAIL	OK	OK	OK	OK
PU-18				OK	OK	OK	OK	OK
PU-19				OK	OK	OK	OK	OK
PU-20				OK	OK	OK	OK	OK
PU-21				FAIL	OK	OK	OK	OK

Cuadro 10.5: Resultados de pruebas de unidad

Cod.	v1.0	v2.1	v3.1	v3.2	v3.3	v3.4
PI-1	OK	OK	OK	OK	OK	OK
PI-2		OK	OK	OK	OK	OK
PI-3		OK	OK	OK	OK	OK
PI-4		OK	OK	OK	OK	OK
PI-5		OK	OK	OK	OK	OK
PI-6		OK	OK	OK	OK	OK
PI-7		OK	OK	OK	OK	OK
PI-8		OK	OK	OK	OK	OK
PI-9		OK	OK	OK	OK	OK
PI-10		OK	OK	OK	OK	OK
PI-11			OK	OK	OK	OK
PI-12			OK	OK	OK	OK
PI-13			OK	OK	OK	OK
PI-14			FAIL	OK	OK	OK
PI-15			OK	OK	OK	OK
PI-16			OK	OK	OK	OK
PI-17			OK	OK	OK	OK
PI-18			OK	OK	OK	OK
PI-19			OK	OK	OK	OK
PI-20			OK	OK	OK	OK
PI-21			OK	OK	OK	OK
PI-22		OK	OK	OK	OK	OK
PI-23		OK	OK	OK	OK	OK
PI-24			OK	OK	OK	OK
PI-25		OK	OK	OK	OK	OK
PI-26			FAIL	FAIL	OK	OK
PI-27		OK	OK	OK	OK	OK
PI-28			OK	OK	OK	OK
PI-29		OK	OK	OK	OK	OK
PI-30			FAIL	OK	OK	OK

Cuadro 10.6: Resultados de pruebas de integración

Cod.	v2.1	v3.3	v3.4
PV-1	OK	OK	OK
PV-2		OK	OK
PV-3		OK	OK
PV-4		OK	OK
PV-5		OK	OK
PV-6		FAIL	OK

Cuadro 10.7: Resultados de pruebas de validación

Parte V

Experimentación y Conclusiones

Capítulo 11

Experimentación

En este capítulo se va a describir un estudio experimental para comprobar que nuestra solución es correcta y que cumple con los objetivos de los que se hablaron al principio de este documento (ver sección 1.2). Esta es a su vez la última fase de cada iteración de la metodología propuesta (ver sección 1.3).

11.1. Descripción del estudio experimental

Para este estudio se van a realizar una serie de experimentaciones, sobre versiones de CPU y GPU de un conjunto de problemas implementados con comunicadores y sin ellos, que nos permitirá comprobar como actúa nuestro modelo, cuáles son las ventajas y desventajas de usarlo y si cumple los objetivos marcados. Se van a realizar los siguientes estudios:

- Esfuerzo en la programación: Se desea medir la dificultad de desarrollar una aplicación utilizando los comunicadores, y compararlo con la dificultad de desarrollar esa misma aplicación sin su uso.
- Esfuerzo de portabilidad: Se desea medir el esfuerzo necesario para convertir un programa que realiza sus cálculos en la CPU a una versión de GPU, o viceversa.
- Rendimiento: Se desea medir el impacto en el rendimiento que tiene nuestro modelo, clasificando así los casos en los que funciona mejor y los casos en los que funciona peor.

Para la obtención del coste de desarrollo se evaluará el código fuente implementado para obtener las versiones experimentales. Para ello se utilizarán unas métricas clásicas para este fin: lineas de código (LOC), el numero de tokens y la complejidad ciclomática [10]. Las dos primeras, las lineas de código y el número de tokens, miden el volumen de código que el programador debe desarrollar. La tercera medida, la complejidad ciclomática, mide la complejidad del esfuerzo racional de programar el problema en términos de ramificaciones en el código y los casos que se deben considerar para desarrollar, probar y depurar dicho programa.

Para conocer el esfuerzo en la portabilidad se compararán el código fuente de las versiones experimentales. Estas comparaciones serán entre versiones de CPU y GPU sin comunicadores y versiones CPU y GPU con comunicadores de los problemas con los que experimentaremos.

Para esto se contará el número de tokens que han sido modificados entre una versión y otra. De ésta forma se obtiene el volumen de código que hay que modificar para cambiar de una versión a otra.

Para obtener los datos sobre el rendimiento de los comunicadores, se deberá obtener el tiempo de ejecución de cada uno de los problemas propuestos en distintas situaciones según el problema. Con la comparación de estos datos se podrá comprobar el overhead producido por añadir los comunicadores como una capa de abstracción extra, obteniendo así una medida de la eficiencia de nuestro modelo para diferentes situaciones.

11.2. Descripción de la máquina de experimentación

Toda la experimentación se realizará sobre una misma máquina. Esta máquina posee dos procesadores Intel Xeon E5-2620v2 con arquitectura de 64 bits, con una frecuencia de reloj de 2.1 GHz y 12 núcleos por procesador (24 núcleos entre los dos). Dispone de 32 GB de memoria RAM. El sistema operativo es un CentOS 7 utilizando la versión del núcleo 3.10.0 compilado para 64 bits.

Utiliza una tarjeta gráfica Nvidia GeForce GTX TITAN Black, con una arquitectura Kepler 3.5. Este dispositivo tiene un total de 2880 núcleos CUDA, que trabajan a una frecuencia de reloj de 980 Mhz, y están organizados en 15 multiprocesadores. Este acelerador dispone de una memoria interna GDDR5 de 6 GB con un ancho de banda de memoria de 336 GB/s. Los drivers utilizados para gestionar esta tarjeta gráfica son los incluidos en la versión 7.5 del SDK de CUDA.

11.3. Casos de estudio

Para este estudio experimental se van a utilizar una serie de problemas que pese a que son muy sencillos, son problemas reales y muy representativos de los problemas en los que se podría aplicar el modelo de comunicadores. Estos problemas son:

- Suma de matrices.
- Multiplicación de matrices.
- PDE solver con el método iterativos de Jacobi.

Para dichos problemas se ha obtenido una versión para la CPU y otra para la GPU implementadas tanto sin el uso de comunicadores, para utilizar como referencia, como utilizando los comunicadores. Por último, la ejecución de los problemas en su la versión de CPU se realizará: con 1 hilo, con 12 y con 24 hilos. Su ejecución con un único hilo nos permite obtener el overhead producido por simular los hilos de GPU en la CPU. La ejecución con 24 hilos (el mismo número que núcleos reales en la CPU de la máquina de experimentación), es para medir el rendimiento general del modelo en la CPU en el caso de máxima explotación de los recursos computacionales de la plataforma. La ejecución con 12 hilos es por el hecho de tener un valor intermedio.

11.3.1. Suma de matrices

La suma de matrices consiste en la suma de dos matrices cuadradas diferentes almacenando el resultado en una tercera matriz: $C_{n\times n} = A_{n\times n} + B_{n\times n}$ (ver Figura. 11.2). En este problema, el cómputo de cada celda no implica ningún tipo de dependencias con el cálculo de otra y además el número de operaciones es muy pequeño. Por este mismo motivo es por el que fue elegido este problema, para ver el comportamiento de los comunicadores cuando el problema es altamente paralelo y presenta bajo coste computacional por cada acceso a las estructuras de datos. Así podemos comprobar si el overhead producido es mayor a los beneficios en este tipo de problemas donde no aparecen otros retrasos impuestos por sincronización o accesos concurrentes. La experimentación se realizará calculando la suma de matrices de distintos tamaños: 128×128, 256×256, 512×512, 1024×1024, 2048×2048 y 4096×4096.

La solución de GPU utilizada para resolver este problema será la suma de vectores presentada en la guía de programación de CUDA [12]. Esto es debido a que en la parte del programa principal, fuera del ámbito de los comunicadores, las matrices se implementarán como vectores al que se le aplican una serie de saltos para obtener un elemento concreto, es decir, se utilizará la función lineal $f(col, fil) = fil * tamFil + col$ donde *tamFil* es el tamaño de una fila, *col* la columna del elemento y *fil* la fila del elemento. La solución de CPU utilizada es la versión clásica de este problema en la que se ha aplicado la misma función lineal para la búsqueda de elementos que la indicada para la parte del programa principal de la solución GPU.

```
1 for(int i = 0; i < ladoMatriz; i++){
2   for(int j = 0; j < ladoMatriz; j++){
3       C[i][j] = A[i][j] + B[i][j];
4   }
5 }
```

Figura 11.1: Pseudocódigo de la suma de matrices.

11.3.2. Multiplicación de matrices

La multiplicación de matrices consiste en el producto de dos matrices cuadradas diferentes almacenando el resultado en una tercera: $C_{n\times n} = A_{n\times n} * B_{n\times n}$ (ver Figura. **??**). En este problema, el resultado de cada celda de la matriz resultante no depende de ningún otro cálculo. Sin embargo, para las diferentes celdas se utilizan elementos de A y B que también son leídos para hacer los cálculos de otras celdas. Se ha elegido este problema debido a su alto coste computacional por cada celda de la matriz, lo cual nos permite medir el comportamiento de lo comunicadores en este tipo de casos y comprobar hasta que punto se llega a notar el overhead producido por su uso. A su vez es un problema más difícil de implementar con diferentes aproximaciones en CPU y GPU, y más difícil de caracterizar para obtener parámetros de ejecución adecuados en GPU. Esto repercute en el coste de programación y se desea medir este hecho. La experimentación se realizará calculando la multiplicación de matrices de distintos tamaños: 128×128, 256×256, 512×512, 1024×1024, 2048×2048 y 4096×4096.

La solución genérica de este problema consiste en resolver la operación $\prod_{k=0}^{n-1} A[i][k] * B[k][j]$ para cada una de las celdas de la matriz C cuyas coordenadas vienen son (i, j).

Para la versión de GPU, esta operación la realiza cada hilo junto con operaciones que nos permiten optimizar los accesos a memoria. Esta versión de referencia es la presentada en la guía de programación de CUDA [12]. La versión de CPU utilizada es la versión clásica con las técnicas de optimización de tiling e inversión de bucles.

```
for(int i = 0; i < ladoMatriz; i++){
  for(int j = 0; j < ladoMatriz; j++){
    for(int k = 0; k < ladoMatriz; k++){
      C[i][j] = A[i][k] + B[k][j];
    }
  }
}
```

Figura 11.2: Pseudocódigo de la suma de matrices.

11.3.3. PDE Jacobi

Este experimento consiste en un programa que resuelve una PDE (Partial Differential Equation) que calcula la transferencia de calor en un espacio discretizado de dos dimensiones representado por una matriz. Para ello resolveremos la ecuación de Poisson [17] discretizada, utilizando el método iterativo de Jacobi [5]. Se ha elegido este experimento debido a que exige una gran cantidad de lanzamientos de kernels, debido a las dependencias de datos entre una iteración y otra, con un esfuerzo computacional por cada celda de la matriz, que no es excesivamente alto lo que nos permite medir como afectan los comunicadores en el coste del lanzamiento de muchos kernels.

La experimentación se realizará calculando la multiplicación de matrices de distintos tamaños: 128×128, 256×256, 512×512 y 1024×1024; y para diferente cantidad de iteraciones: 1, 10 y 100. Esto nos da una medida de cómo crece el coste acumulado al aumentar el número de lanzamientos de kernels. Además, al aumentar el tamaño se puede ver el efecto con distintos costes de computación por iteración.

La solución de CPU es una versión paralelizada mediante OpenMP de la versión secuencial mostrada en la Figura 11.3. En esta solución por cada iteración del autómata se hace una copia de la matriz y después se actualiza la matriz original con la media de sus vecinos: superior, inferior y laterales. La solución de GPU es similar a la propuesta para la CPU.

11.4. Resultados de la experimentación

A continuación, presentamos los resultados de la experimentación realizada con el prototipo para cada uno de los experimentos propuestos en cada uno de los casos de estudio.

11.4.1. Esfuerzo en la programación

A través de una evaluación del propio código fuente implementado para obtener las versiones experimentales se obtienen los valores que se muestran en el Cuadro 11.1. Se puede observar que los comunicadores reducen siempre la complejidad ciclomática, esto significa que utilizando los comunicadores hay menos puntos donde el programador tiene que pensar

```
1  // BUCLE ITERACIONES
2  for(int i = 0; i < iteraciones; i++){
3    // GUARDAR COPIA DE LOS DATOS
4    for(int j = 1; j < ladoMatriz-1; j++){
5      for(int k = 1; k < ladoMatriz-1; k++){
6        matrizCopia[j][k] = matriz[j][k];
7      }
8    }
9    // ACTUALIZAR CADA CELDA DE LA MATRIZ
10   for(int j = 1; j < ladoMatriz-1; j++){
11     for(int k = 1; k < ladoMatriz-1; k++){
12       // MEDIA DE LOS CUATRO VECINOS
13       matriz[j][k] = matrizCopia[j][k-1] + matrizCopia[j][k+1];
14       matriz[j][k] += matrizCopia[j-1][k] + matrizCopia[j+1][k];
15       matriz[j][k] /= 4;
16     }
17   }
18 }
```

Figura 11.3: Pseudocódigo PDE de Jacobi en secuencial.

en operaciones complejas, aumentando la velocidad de desarrollo y reduciendo los posibles errores. Además, hay que añadir que en ciertas ocasiones el número de lineas de código y el número de tokens aumentan, esto se produce por las macros de caracterización, que aunque simplifican el uso de los aceleradores, evitando al programador múltiples ejecuciones de prueba y error para localizar los valores adecuados de los parámetros de ejecución (tamaño de bloques de hilos, o tamaño de caché variable), aumenta el número de líneas.

A partir de estos resultados se puede decir que en general el modelo aumenta el número de tokens/lineas con respecto a OpenMP, ya que OpenMP está especialmente diseñado para hilos en CPU, mientras que nuestra aproximación intenta asemejar el uso de una GPU para reducir el esfuerzo de portabilidad. En el caso de GPUs depende de la complejidad del programa. Se mejora en casos más complejos como la multiplicación de matrices por que los comunicadores exigen una serie de operaciones previas de inicialización, binding, etc. que implican un coste constante en líneas de código y tokens, que se amortiza según crece el programa.

11.4.2. Esfuerzo en la portabilidad

En el Cuadro 11.2 se puede ver el coste, en número de tokens, de pasar estos problemas de la versión de CPU a la de GPU y viceversa. Según los resultados mostrados, en los comunicadores apenas hay que realizar cambios en el código que nos permita ejecutar en otra plataforma. Sin embargo, hay que tener el cuenta que el hecho de hacer optimizaciones específicas de la plataforma puede aumentar este coste, pero siempre siendo menor que sin el uso de comunicadores. Por ejemplo, la multiplicación de matrices donde el número de tokens mostrado para comunicadores ya incluye optimizaciones diferentes con cambios substanciales, en los kernel de CPU y GPU.

Caso de estudio	Versión	Líneas de código	Nº Tokens	Complejidad Ciclomática
Suma matrices	OpenMP	49	393	10
	Comm. CPU	60	528	9
	CUDA	61	448	10
	Comm. GPU	66	541	9
Multiplicación de Matrices	OpenMP	59	528	15
	Comm. CPU	71	660	13
	CUDA	123	974	16
	Comm. GPU	83	710	12
PDE Jacobi	OpenMP	52	461	17
	Comm. CPU	69	592	15
	CUDA	67	574	19
	Comm. GPU	76	623	15

Cuadro 11.1: Esfuerzo en el desarrollo

Caso de estudio	CUDA - OpenMP	Comm. CPU - Comm. GPU
Suma matrices	55	29
Multiplicación matrices	92	30
PDE Jacobi	31	10

Cuadro 11.2: Esfuerzo de portar el algoritmo

11.4.3. Rendimiento

En la Figuras 11.4 y 11.5 se puede ver la diferencia, en tanto por ciento, del tiempo de ejecución entre el uso de comunicadores y sin ellos, para los problemas de suma de matrices y multiplicación de matrices en diferentes tamaños. Como se puede observar, cuanto mayor es el tamaño, menor es la diferencia. El motivo es que el sobrecoste por simular los hilos de GPU, en el caso de las CPUs, y el enlazado/desenlazado de variables, en todos los casos, se ve compensado al crecer la carga computacional. Los resultados de la suma de matrices se encuentra en el Cuadro 11.3 y el de la multiplicación de matrices en el Cuadro 11.4.

Para casos como la PDE resuelta utilizando el método de Jacobi, al igual que la suma de matrices y la multiplicación de matrices, se observa una tendencia a reducir la diferencia en el tiempo de ejecución entre la versión de referencia y la versión con comunicadores (ver Figuras 11.6 y 11.7) al crecer el tamaño del problema. Además se puede apreciar que el overhead por añadir las tareas a la lista de tareas a ejecutar, se ve diluido por el número de iteraciones. Esto se debe a que el lanzamiento asíncrono de los kernels hace que solape la computación de una iteración con el hecho de añadir una tarea a la cola. Los resultados de esta experimentación para este caso de estudio se puede ver en los Cuadros 11.5 y 11.6.

11.4.4. Resumen de los resultados

Los resultados obtenidos indican que el uso de comunicadores exige menos esfuerzo de razonamiento que la programación con los modelos nativos de cada tipo de dispositivo considerado. Aunque implica un mayor esfuerzo de programación que con OpenMP, aproxima

Dispositivo	Tamaño de la matriz	Tiempo Referencia	Tiempo Comm.	Diferencia (%)
CPU (1 hilo)	128 x 128	0,0000996	0,0002902	191,36
	256 x 256	0,0003451	0,0005666	64,19
	512 x 512	0,0013397	0,0015238	13,74
	1024 x 1024	0,0051232	0,0054112	5,62
	2048 x 2048	0,0203928	0,0212950	4,42
	4096 x 4096	0,0811645	0,0841435	3,67
CPU (12 hilos)	128 x 128	0,0003532	0,0009327	164,08
	256 x 256	0,0003789	0,0007046	85,96
	512 x 512	0,0006299	0,0010187	61,71
	1024 x 1024	0,0019041	0,0023109	21,37
	2048 x 2048	0,0104424	0,0108903	4,29
	4096 x 4096	0,0437742	0,0444249	1,49
CPU (24 hilos)	128 x 128	0,0026298	0,0040915	55,58
	256 x 256	0,0027008	0,0033305	23,32
	512 x 512	0,0029455	0,0030558	3,74
	1024 x 1024	0,0042234	0,0044201	4,66
	2048 x 2048	0,0127403	0,0131911	3,54
	4096 x 4096	0,0464191	0,0469281	1,10
GPU	128 x 128	0,0004723	0,0007556	59,99
	256 x 256	0,0007365	0,0010357	40,63
	512 x 512	0,0022368	0,0026413	18,08
	1024 x 1024	0,0061153	0,0067909	11,05
	2048 x 2048	0,0235840	0,0250574	6,25
	4096 x 4096	0,0898446	0,0950674	5,81

Cuadro 11.3: Rendimiento de la suma de matrices

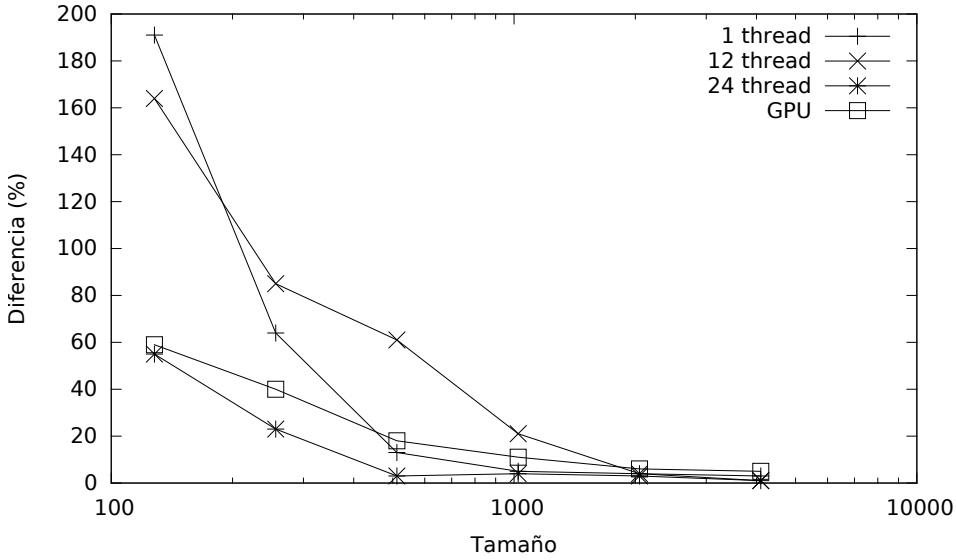

Figura 11.4: Gráfico de diferencia en tiempos de ejecución de la suma de matrices.

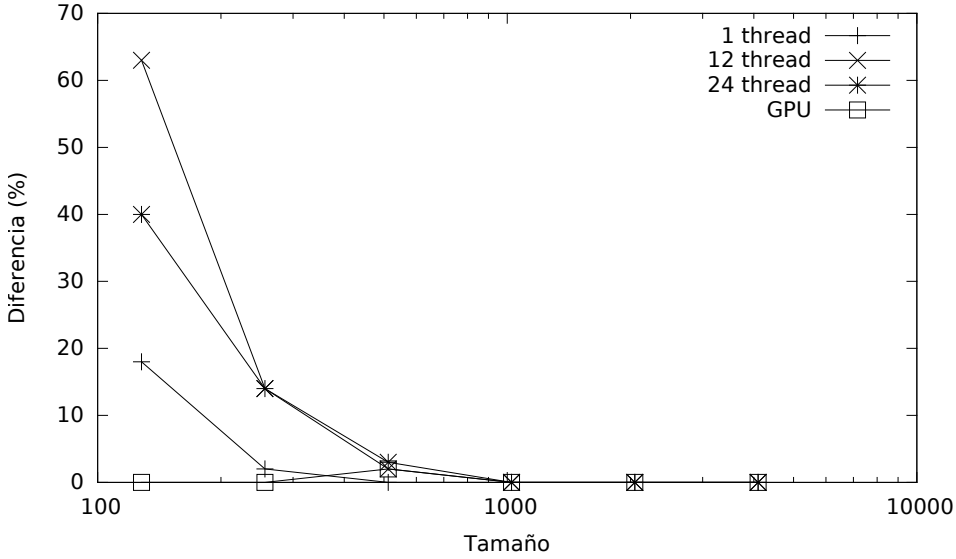

Figura 11.5: Gráfico de diferencia en tiempos de ejecución de la multiplicación de matrices.

Dispositivo	Tamaño de la matriz	Tiempo Referencia	Tiempo Comm.	Diferencia (%)
CPU (1 hilo)	128 x 128	0,0033889	0,0040179	18,56
	256 x 256	0,0335536	3,5197400	0,51
	512 x 512	0,2957690	0,2961090	0,11
	1024 x 1024	3,5019400	3,5197400	0,51
	2048 x 2048	28,0057000	28,1083000	0,37
	4096 x 4096	225,4950000	225,6050000	0,05
CPU (12 hilos)	128 x 128	0,0008352	0,0013621	63,07
	256 x 256	0,0032398	0,0037215	14,87
	512 x 512	0,0250494	0,0257355	2,74
	1024 x 1024	0,2952000	0,2952490	0,02
	2048 x 2048	2,3454000	2,3504800	0,22
	4096 x 4096	18,8732000	18,9812000	0,57
CPU (24 hilos)	128 x 128	0,0029815	0,0041956	40,72
	256 x 256	0,0055678	0,0064006	14,96
	512 x 512	0,0223375	0,0230440	3,16
	1024 x 1024	0,2159780	0,2169330	0,44
	2048 x 2048	1,6850000	1,6931700	0,48
	4096 x 4096	15,0587000	15,1123000	0,36
GPU	128 x 128	0,7196500	0,7227200	0,43
	256 x 256	0,7215310	0,7221470	0,09
	512 x 512	0,7186480	0,7330650	2,01
	1024 x 1024	0,7323710	0,7376000	0,71
	2048 x 2048	0,7928690	0,7933010	0,05
	4096 x 4096	1,2247300	1,2256536	0,08

Cuadro 11.4: Rendimiento de la multiplicación de matrices

Dispositivo	Tamaño de la matriz	Nº iter.	Tiempo Referencia	Tiempo Comm.	Diferencia (%)
CPU (1 hilo)	128 x 128	1	0,0002244	0,0004044	80,21
		10	0,0019424	0,002400	23,60
		100	0,0180312	0,0194137	7,67
	256 x 256	1	0,0008305	0,0013330	60,50
		10	0,0079720	0,0109796	37,73
		100	0,0870684	0,1019100	17,05
	512 x 512	1	0,0033223	0,0047777	43,81
		10	0,0413018	0,0463003	12,10
		100	0,4045620	0,4473590	10,58
	1024 x 1024	1	0,0222708	0,0275608	23,75
		10	0,2620800	0,2862230	9,21
		100	2,7045700	2,8428300	5,11
CPU (12 hilos)	128 x 128	1	0,0008903	0,0012513	40,55
		10	0,0032344	0,0043841	35,55
		100	0,0235542	0,0280477	19,08
	256 x 256	1	0,0008265	0,0011226	35,83
		10	0,0036476	0,0039811	9,14
		100	0,0196138	0,0201403	2,68
	512 x 512	1	0,0009241	0,0011653	26,10
		10	0,0067511	0,0077545	14,86
		100	0,0594238	0,0627500	5,60
	1024 x 1024	1	0,0033296	0,0045571	36,86
		10	0,0250861	0,0288569	15,03
		100	0,2074530	0,2092000	0,84
CPU (24 hilos)	128 x 128	1	0,0023446	0,0046476	98,23
		10	0,0053335	0,0112543	111,01
		100	0,0112401	0,0512582	356,03
	256 x 256	1	0,0025234	0,0041203	63,29
		10	0,0058942	0,0129409	119,55
		100	0,0151173	0,0691735	357,58
	512 x 512	1	0,0029055	0,0066379	128,46
		10	0,0078092	0,0218052	179,23
		100	0,0283107	0,1375220	385,76
	1024 x 1024	1	0,0054966	0,0128890	134,49
		10	0,0160416	0,0567235	253,60
		100	0,0820176	0,4128800	403,40

Cuadro 11.5: Rendimiento de la PDE Jacobi.

Dispositivo	Tamaño de la matriz	Nº iter.	Tiempo Referencia	Tiempo Comm.	Diferencia (%)
GPU	128 x 128	1	0,0004162	0,0005051	21,36
		10	0,0009242	0,0010869	17,60
		100	0,0023224	0,0024373	4,95
	256 x 256	1	0,0005974	0,0007397	23,81
		10	0,0013180	0,0015058	14,25
		100	0,0032952	0,0034277	4,02
	512 x 512	1	0,0015250	0,0017057	11,85
		10	0,0033846	0,0034346	1,48
		100	0,0088434	0,0089276	0,95
	1024 x 1024	1	0,0040746	0,0041943	2,94
		10	0,0092553	0,0094057	1,63
		100	0,0255110	0,0255500	0,15

Cuadro 11.6: Rendimiento de la PDE Jacobi.(cont)

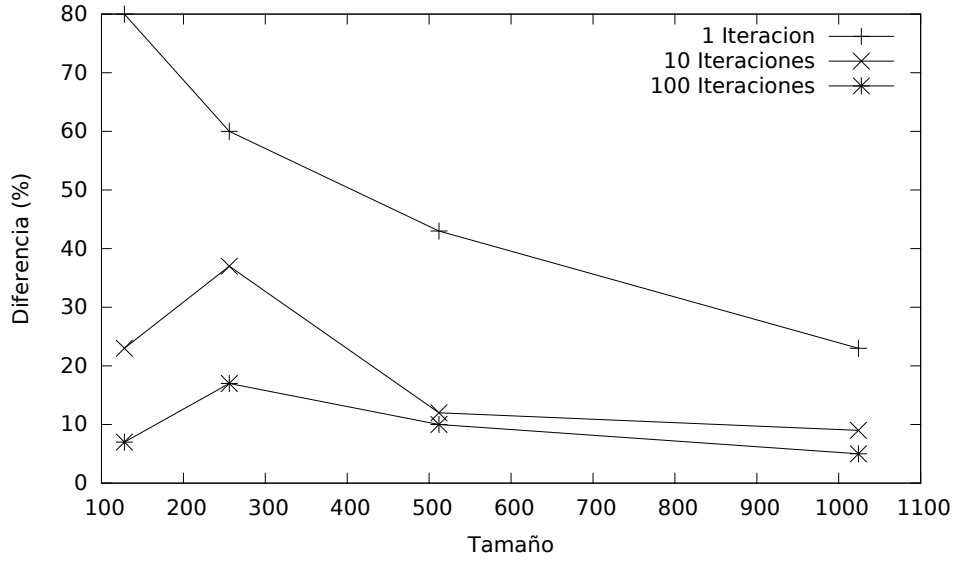

Figura 11.6: Gráfico de diferencia en tiempos de ejecución del Jacobi en la CPU.

la programación de CPUs y GPUs de tal forma, que el esfuerzo de portabilidad entre ambas plataformas se minimiza. El esfuerzo extra de introducir creaciones e inicializaciones de comunicadores se va diluyendo al crecer la complejidad de los programas principales. En cuanto a rendimiento, los overheads introducidos también se van reduciendo al aumentar los tamaños o repeticiones del problema.

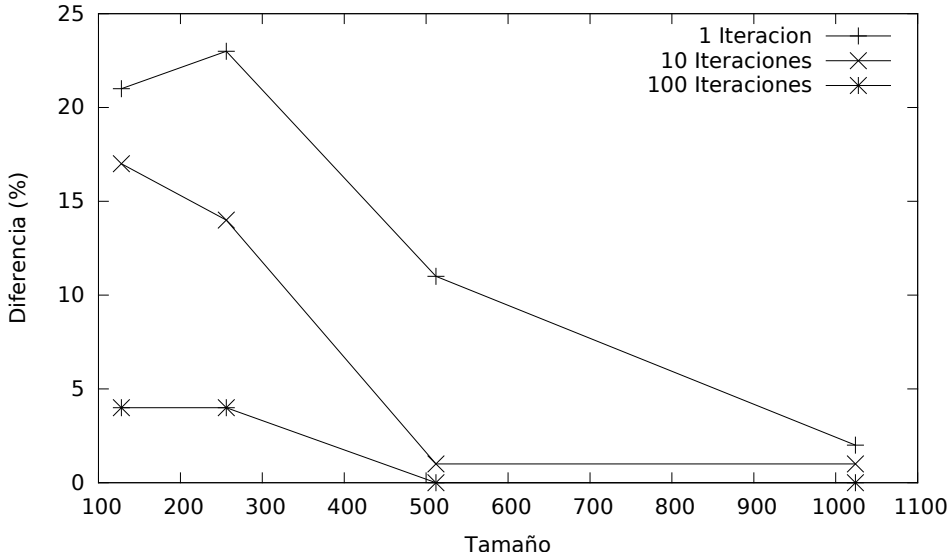

Figura 11.7: Gráfico de diferencia en tiempos de ejecución del Jacobi en la GPU.

Capítulo 12

Conclusiones y Trabajo futuro

En este proyecto se ha propuesto el modelo de comunicadores, un modelo de programación que simplifica la codificación de aplicaciones para sistemas heterogéneos. Éste está basado en el concepto de comunicador, una entidad abstracta que administra y lanza secuencias de kernels en aceleradores hardware o en núcleos del procesador principal. Este modelo provee de mecanismos para:

- Asociar comunicadores a dispositivos.
- Definir kernels fácilmente portables a diferentes tipos de dispositivos.
- Seleccionar valores apropiados para el lanzamiento de kernels en diferentes dispositivos a través de su caracterización.
- Transferir los datos entre los espacios de memoria del host y del dispositivo cuando sean necesarios.

Este modelo homogeneiza la programación y la administración de los kernels, acercando la programación multihilo a la programación de aceleradores, teniendo en cuenta las diferencias entre las plataformas aceleradoras para tener un buen rendimiento.

Nuestro estudio experimental ha demostrado que realmente el usar comunicadores reduce el coste de la programación al aumentar la complejidad de los programas, reduciendo el coste de portabilidad entre diferentes dispositivos, y que el overhead producido por la biblioteca, es compensado cuando la carga computacional, los tamaños del problema, o la cantidad de repeticiones de ejecución de los kernels crece.

12.1. Objetivos cumplidos

Durante la introducción se explicaron los objetivos que nos hemos marcado para este proyecto. Dichos objetivos son:

- Configuración automática del dispositivo.
- Transferencias transparentes al programador.
- Unificar distintos paradigmas de programación.

Tras completar este trabajo, se dispone de un modelo y de un prototipo, los cuales cumplen con estos objetivos. Además de estos objetivos principales, se han ido cumpliendo una serie de objetivos esperados de otra naturaleza, de entre los que cabe destacar la elaboración y presentación en eventos científicos de dos artículos. Uno de ellos se ha presentado en las Jornadas de Paralelismo celebradas en Córdoba durante el mes des septiembre de 2015. El otro, fue presentado en enero de 2016 en Praga en el workshop HLPGPU celebrado dentro de la conferencia HiPEAC 2016.

12.2. Conocimientos adquiridos

Durante el desarrollo de este trabajo, se han necesitado y adquirido una serie de conocimientos. Primero, se han obtenido una serie de conocimiento técnicos sobre la computación de alto rendimiento, la programación paralela y programación de aceleradores hardware entre otros. Además se han obtenido una serie de conocimiento en cuanto al mundo de la investigación científica, como las metodologías utilizadas y cómo se redacta un artículo, y el esfuerzo que hay detrás de ello, entre otros. También ha servido para entender la importancia de una buena gestión de proyecto y lo difícil que es en ocasiones realizar una buena planificación y cumplir los plazos establecidos. Por último, ha sido una experiencia en el desarrollo de un proyecto completo con su documentación asociada y siguiendo una metodología concreta elegida previamente.

12.3. Trabajo futuro

Pese al término de este proyecto, no significa que las posibilidades que nos ofrece el modelo de los comunicadores estén totalmente exploraras, así como las tecnologías que se pueden aplicar para mejorar su rendimiento. Como trabajo futuro, se han pensado una serie de posibilidades:

- Mejoras en el rendimiento del prototipo:
  - Buscar una solución para reducir el overhead producido por la emulación de los hilos de una GPU en una CPU.
  - Optimizar los bucles que emulan los hilos de las GPU.
  - Usar tablas hash para sustituir las listas de variables enlazadas e internas, mejorando así el rendimiento en las búsquedas de tiles.
  - Uso de comunicaciones asíncronas.
- Estudiar el efecto de distintas políticas de planificación para un planificador más completo.
- Implementar versiones del prototipo para otros tipos de aceleradores hardware, como por ejemplo la XeonPhi cuya solución podría ser muy parecida a la de CPU, puesto que su paradigma de programación es muy similar.

Parte VI

Apéndices y Bibliografía

Apéndice A

Contenido del CD-ROM

La memoria presente contiene adjunto un CD-ROM que incluye toda la información detallada del proyecto. En esta sección explicaremos el contenido de dicho CD-ROM y como está organizado. El CD-ROM contendrá el código utilizado en el desarrollo del sistema, tanto el nuevo prototipo, como los casos de estudio, además contiene una copia digital de la memoria.

A.1. Árbol de directorios

El directorio raiz del cd-rom que acompaña esta memoria tiene la siguiente estructura:

- **memoria.pdf**: Versión digital de esta misma memoria.
- **fuentes**: Directorio con el código fuente del prototipo junto con los casos de estudio.
  - **CMakeList.txt**: Fichero usado por cmake para crear el fichero make. Este en concreto es el raiz de todos los ficheros CMakeList.txt de todo el proyecto.
  - **cal**: Es el directorio que contiene los fuentes de la biblioteca del proyecto.
    - **CMakeList.txt**: Fichero usado por cmake para crear el fichero make. Este en concreto es el fichero encargado de generar la biblioteca del proyecto.
    - **inc**: Contiene los ficheros de cabera públicos de la biblioteca.
    - **src**: Contiene el código fuente de la biblioteca.
    - **doc**: Contiene los archivos de configuración del generador de documentación doxygen.
  - **examples**: Este directorio contiene ejemplos de uso de la biblioteca utilizados como casos de estudio. Cada ejemplo tiene un directorio propio con el nombre del ejemplo que contiene lo siguiente:
    - **CMakeList.txt**: Fichero usado por cmake para crear el fichero make. Este en concreto es el raiz de todos los ficheros CMakeList.txt de todo el proyecto.
    - **src**: Contiene el código fuente del ejemplo.
- **articulos**: Contiene la versión digital de los artículos presentados en las Jornadas de paralelismo y el workshop HLPGPU celebrado dentro de la conferencia HiPEAC.

  - **jornadas.pdf**: Versión digital del artículo presentado en las Jornadas de paralelismo.
  - **hlpgpu.pdf**: Versión digital del artículo presentado en el workshop HLPGPU.

- **binarios:** Este directorio contiene los binarios compilados del código fuente descrito anteriormente.
  - **cal**: Contiene el binario de la biblioteca.
  - **examples**: Contiene los binarios de los ejemplos.

Bibliografía

[1] W. R. Adrion. Research Methodology in Software Engineering. Summary of the Dagstuhl Workshop on Future Directions in Software Engineering. *SIGSOFT Software Engineering Notes*, 18(1):36–37, 1993.

[2] Q.-k. Chen and J.-k. Zhang. A Stream Processor Cluster Architecture Model with the Hybrid Technology of MPI and CUDA. In *ICISE'2009*, pages 86–89, dec. 2009.

[3] O. Consortium. OpenACC: Directives for Accelerators, 2011–2015. on `http://www.openacc-standard.org/`.

[4] O. Consortium. OpenMP 4.0 Specifications, July 2013. on `http://openmp.org/wp/openmp-specifications/`.

[5] G. H. Golub and C. F. van Loan. *Matrix Computations*, chapter 8. Johns Hopkins University Press, 1996.

[6] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D. R. Llanos. An extensible system for multilevel automatic data partition and mapping. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1145–1154, 2014.

[7] T. D. Han and T. S. Abdelrahman. hiCUDA: a high-level directive-based language for GPU programming. In D. R. Kaeli and M. Leeser, editors, *GPGPU*, volume 383, pages 52–61. ACM, 2009.

[8] N. Karunadasa and D. Ranasinghe. Accelerating high performance applications with CUDA and MPI. In *ICIIS'2009*, pages 331–336, dec. 2009.

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

[10] T. J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[11] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.1. Technical report, University of Tennessee, 2015.

[12] NVIDIA. NVIDIA CUDA C Programming Guide 7.5, 2015. Last visit: November 16th, 2015.

[13] H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. Optimizing an APSP implementation for NVIDIA GPUs using kernel characterization criteria. *The Journal of Supercomputing*, 70(2):786–798, 2014.

[14] H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. Tuccompi: A multi-layer model for distributed heterogeneous computing with tuning capabilities. *International Journal of Parallel Programming*, 43(5):939–960, 2015.

[15] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano. A tuned, concurrent-kernel approach to speed up the apsp problem. In *The 13th International Conference Computational and Mathematical Methods in Science and Engineering (CMMSE 2013)*, volume 4, pages 1114–1125, Almería, España, Junio 2013.

[16] R. S. Pressman. *Software Engineering. A Practitioner's Approach.* The McGraw-Hill Companies, Inc.„ 1221 Avenue of the Americas, New York, NY 10020, 7th edition, 2010.

[17] R. D. Richtmyer. *Principles of Advanced Mathematical Physics*, volume 1, chapter 6. Springer-Verlag New York Inc., 1978.

[18] I. Sommerville. *Software Engineering.* Pearson Education Limited, United Kingdom, 7th edition, 2004.

[19] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering*, 12(3):66–73, May 2010.

[20] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In J. N. Amaral, editor, *LCPC'2008*, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms.* The Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 2nd edition, 2001.

[22] TOP500.org. Top500 Supercomputing Sites, Nov 2014. on `http://www.top500.org/`.

[23] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. ubench: exposing the impact of cuda block geometry in terms of performance. *The Journal of Supercomputing*, 65(3):1150–1163, 2013.