View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by UPCommons. Portal del coneixement obert de la UPC

Títol: OmpSsCL:
Entorno para la programación automática
de aplicaciones OpenCL

Autor: Filgueras Izquierdo, Antonio

Data: 14 de Junio

Director: Jiménez González, Daniel

Departament del director:
Departament d'Arquitectura de Computadors

Titulació: Enginyeria Informàtica

Centre: Facultat d'Informàtica de Barcelona (FIB)

Universitat: Universitat Politècnica de Catalunya (UPC)
BarcelonaTech

DADES DEL PROJECTE

Títol del Projecte:	OmpSsCL: Entorno para la programación automática de aplicaciones OpenCL
Nom de l'estudiant:	Filgueras Izquierdo, Antonio
Titulació:	Enginyeria en Informàtica
Crèdits:	37.5
Director/Ponent:	Jiménez González, Daniel
Departament:	Departament d'Arquitectura de Computadors

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President: Álvarez Martínez, Carlos

Vocal: Oliveras Llunell, Albert

Secretari: Jiménez González, Daniel

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:

Data:

Universitat Politècnica de Catalunya - Facultat d'Informàtica de Barcelona
Enginyeria Informàtica

OmpSsCL: Entorno para la programación automática de aplicaciones OpenCL

Departament d'Arquitectura de Computadors

Director: Jiménez González, Daniel
Alumno: Filgueras Izquierdo, Antonio

Barcelona, 2012

Agradecimientos

A Moisés Guillén, por el gran trabajo realizado y su ayuda prestada.

A Daniel Jimenez, por su inestimable ayuda, este proyecto no habría sido posible sin él.

Índice general

Índice de figuras 5

Índice de tablas 8

Índice de códigos 9

Glosario 11

1. Introducción 13

1.1. Objetivos 14

1.1.1. Objetivos específicos 15

1.2. Metodología 16

1.3. Organización de la memoria 18

2. OpenMP 20

3. OmpSs 23

3.1. Nanos 25

4. OpenCL 26

4.1. API OpenCL 27

5. Mercurium 35

6. Diseño y funcionamiento **37**

7. Implementación **43**

7.1. Transformación de código 43

7.1.1. Generación de código OpenCL 45

7.1.2. Generación de código del host 48

7.2. API del Runtime 52

7.2.1. Tipos y estructuras de datos 52

7.2.2. Procedimientos 53

7.3. Gestión de recursos 56

7.4. Optimización de transferencias de memoria 61

8. Instalación y uso **62**

8.1. Requisitos del sistema 62

8.2. Instalación 63

8.2.1. Instalación de los requisitos 63

8.3. Uso del sistema 69

8.3.1. Opciones del compilador 69

9. Resultados **72**

9.1. Entorno de pruebas 72

9.2. Filtro FIR 73

9.2.1. Análisis de la implementación OmpSsCL 75

9.3. K-means 77

9.3.1. Análisis de la implementación básica OmpSsCL 79

9.3.2. Análisis de la implementación OmpSsCL + present 82

9.4. BFS 85

9.4.1. Análisis de la implementación básica usando OmpSsCL 87

9.4.2. Análisis de la implementación OmpSsCL usando present (W) . . . 90

9.4.3. Análisis de la implementación OmpSsCL usando present (W+R) . 93
9.4.4. Análisis de la implementación OmpSsCL con present + flush . . . 97
9.5. Comparativa de rendimiento 100
9.6. Pruebas de ejecución concurrente 103
9.6.1. Ejecución concurrente en múltiples dispositivos 103
9.6.2. Control de dependencias 104

10. Planificación y costes **105**

10.1. Costes del proyecto 105
10.1.1. Resumen de costes 106
10.2. Planificación 106

11. Conclusiones **108**

I Apéndices **110**

A. Información del sistema **111**

A.1. Resultados de la ejecución de CLInfo 111
A.2. Contenido /proc/cpuinfo 116

Índice de figuras

1.1. Proceso de desarrollo. 18

2.1. Modelo fork-join. 21

4.1. Diagrama de capas de OpenCL. 27

4.2. Modelo de datos de OpenCL. 30

5.1. Diagrama de fases de Mercurium. 36

6.1. Diagrama del funcionamiento de OmpSsCL. 40

6.2. Diagrama de capas de una aplicación OmpSsCL. 41

7.1. Ejemplos de rangos de datos a copiar a un dispositivo. 60

9.1. Tiempo de ejecución de las diferentes implementaciones del filtro FIR. . . 74

9.2. *Zoom* de la traza de ejecución de la implementación de FIR con OmpSsCL. 76

9.3. Leyenda de las trazas de ejecución. 76

9.4. Porcentaje de tiempos de FIR empleado en el *runtime* de OmpSsCL. . . . 77

9.5. Tiempo de ejecución de las diferentes implementaciones de k-means. . . . 78

9.6. *Zoom* de una traza de ejcución de la omplementación básica de k-means con OmpSsCL. 81

9.7. Porcentajes del tiempo de la implementación básica de k-means empleado en el runtime de OmpSsCL. 82

9.8. *Zoom* de una traza de ejecución de k-means en OmpSsCL con la cláusula `present`. 83
9.9. Comparativa de tiempos de ejecución (arriba) y del porcentaje en los diferentes estados de ejecución (abajo) k-means usando o no la cláusula `present`. 84
9.10. Comparativa de tiempos de las diferentes versiones de BFS. 86
9.11. *Zoom* de la traza de ejecución de la versión sin optimizaciones de BFS. . . 90
9.12. Fracción de tiempo de ejecución en los diferentes estados del runtime OmpSsCL de la implementación BFS con OmpSsCL. 90
9.13. *Zoom* de la traza de ejecución de la implementación de BFS con optimizaciones en la escritura de datos (present (W)). 92
9.14. Comparativa de tiempos de ejecución (arriba) y del porcentaje en los diferentes estados de ejecución (abajo) de BFS usando o no la cláusula `present` para las variables de escritura (OmpSsCL y OmpSsCL present (W)). . . . 93
9.15. *Zoom* de la traza de ejecución de BFS con optimizaciones en la lectura y escritura de datos (present (W+R)). 96
9.16. Comparativa de tiempos de ejecución (arriba) y del porcentaje en los diferentes estados de ejecución (abajo) de BFS usando o no la cláusula `present` para las variables lectura y escritura (OmpSsCL present (W) y OmpSsCL present (W+R)). 97
9.17. *Zoom* de la traza de ejecución de la implementación de BFS utilizando la cláusula `flush`. 99
9.18. Comparativa de tiempos de ejecución (arriba) y del porcentaje en los diferentes estados de ejecución (abajo) de BFS usando o no la construcción `flush` (OmpSsCL present (W+R) y OmpSsCL present + flush). 100
9.19. Comparativa del *speedup* de las aplicaciones programadas con OmpSsCL y OpenCL respecto a la ejecución secuencial en el procesador del *host*. . . 101
9.20. *Zoom* de la traza de ejecución mostrando la ejecución concurrente de tareas.103

9.21. Leyenda de las trazas de ejecución para dispositivos. 104
9.22. Zoom de una traza de ejecución mostrando la gestión de dependencias entre tareas. 104

10.1. Planificación de las tareas 107

Índice de tablas

9.1. Características del sistema de pruebas. 73

9.2. Características de la GPU. 73

9.3. Tiempo de cálculo (*elapsed*) de las diferentes implementaciones del filtro FIR. 75

9.4. Tiempos de ejecución de las diferentes implementaciones de k-means . . . 78

9.5. Tiempos de ejecución de las diferentes implementaciones de BFS. 85

9.6. Tamaño de los diferentes datos de entrada de las aplicacaiones. 101

9.7. Tiempos de las diferentes aplicaciones e implementaciones de éstas, para los diferentes conjuntos de datos A, B y C. 102

10.1. Coste de los recursos humanos del proyecto. 106

10.2. Coste de los materiales amortizables del proyecto. 106

10.3. Resumen de costes del proyecto. 106

Índice de códigos

2.1. Ejemplo de programa OpenMP. 21
3.1. Ejemplo de programa OmpSs. 24
4.1. Código del *host* de la suma de vectores implementada con OpenCL. . . . 31
4.2. Código del *kernel* OpenCL de la suma de vectores. 33
6.1. Ejemplo de código de entrada escrito para OmpSsCL. 41
7.1. Ejemplo de código de entrada para la suma de dos vectores para OmpSsCL. 44
7.2. Código OpenCL generado para implementar el kernel de la suma de vectores. 45
7.3. Ejemplo de código con un parámetro con un rango de valores `[B:E]`. . . . 46
7.4. Ejemplo de código OpenCL con desplazamientos. 47
7.5. Código del *host* generado automáticamente para la inicialización del entorno y las tareas expresadas en OmpSsCL. 49
7.6. Código del host que invoca al *kernel* OpenCL. 50
7.7. Implementación de un filtro FIR usando OmpSsCL. 57
7.8. Código *kernel* OpenCL generado a partir del código 7.7. 59
7.9. Bucle de ejemplo para determinar los rangos de datos que se deben copiar al dispositivo. 60
8.1. Ejemplo de configuración del Makefile de OpenCLNanos 68
9.1. Implementación filtro FIR en OmpSsCL. 75
9.2. Implementación básica de k-means en OmpSsCL. 80
9.3. Código de llamada a la tareas `kmeans_task`. 81

9.4. Implementación OmpSsCL de k-means usando la cláusula `present`. 82

9.5. Definición de las tareas para la implementación con OmpSsCL para BFS. 87

9.6. Código OmpSsCL de llamada a las tareas de BFS. 88

9.7. Código OmpSsCL de las tareas de BFS utilizando la cláusula present para las variables de entrada (versión BFS present (w)). 91

9.8. Código OmpSsCL de las tareas de BFS utilizando la cláusula present para las variables de entrada (versión BFS present (W+R)). 94

9.9. Código de llamada a las tareas de la versión OmpSsCL present+flush de BFS. 98

A.1. Información del entorno OpenCL. 111

A.2. Información del procesador. 116

Glosario

Application Programming Interface (API)

(Interfaz de programación de Aplicaciones) Especificación (o conjunto de especificaciones), típicamente basadas en código fuente, que permiten y facilitan la interacción de distintos componentes software.

Benchmark

Conjunto de herramientas que permiten evaluar el rendimiento de un sistema.

C

Lenguaje de programación imperativo y de propósito especifico.

C++

Lenguaje de programación orientado a objetos basado en *C*.

Central Processing Unit (CPU)

Unidad central de proceso o procesador. Es el componente de ejecutar las instrucciones que componen un programa.

Fortran

Lenguaje de programación imperativo, orientado al calculo numérico y computación científica, desarrollado inicialmente por *IBM*.

Framework

Colección de elementos de software que definen una *API* y que proveen una serie de funcionalidades en base a las cuales otro software puede ser desarrollado.

Graphics Processing Unit (GPU)

Dispositivo destinado principalmente al procesamiento de gráficos.

Host

Sistema de propósito general que ejecuta el programa principal y tiene acceso a los dispositivos.

Kernel (OpenCL)

Función escrita en lenguaje OpenCL, basado en C99, que se ejecuta en un dispositivo.

Paraver

Sistema de visualización y análisis de trazas de ejecución desarrollado por el *Barcelona Supercomputing Center*.

Runtime (librería)

Programa en forma de librería destinado a la implementación de una serie de funcionalidades durante la ejecución del programa.

Thread (Hilo de ejecución)

Entidad de procesamiento que permite a un proceso ejecutar concurrentemente varias tareas o aprovechar diversas *CPU* (paralelismo).

Capítulo 1

Introducción

La tendencia generalizada desde los inicios de la informática ha sido la mejora de la tecnología (ley de Moore [8]), el aumento de la complejidad de los *chips* y el incremento de la frecuencia de funcionamiento de los procesadores. Todo esto provocó un aumento del consumo en estos dispositivos que se ha ido incrementando hasta aproximadamente el año 2004.

A partir de este año, la tendencia ha sido incorporar unidades de procesamiento menos complejas y con menor frecuencia en un mismo *chip*, y con tal de no sacrificar rendimiento, se agrupan varias de estas unidades de procesamiento más simples en un solo dispositivo. Con este mayor número de unidades de procesamiento se intenta aprovechar la mejora tecnológica (más transistores en un mismo *chip*) y mejorar el rendimiento potencial de los *chips* sin aumentar considerablemente el consumo energético.

Por otro lado, la demanda de aplicaciones cada vez más complejas, ha dado lugar a la aparición de aceleradores como los procesadores gráficos o *GPU*. Estos dispositivos, por su naturaleza, agrupan una buen número de núcleos de procesamiento, proporcionando una gran potencia de cálculo a la vez que se contiene el consumo energético.

La gran potencia de cálculo de las *GPU*, en un principio diseñadas únicamente para la representación de gráficos complejos, ha motivado la aparición de diferentes *frameworks* (como CUDA, de NVIDIA o StreamSDK de AMD/ATi) para poder programar estos dispositivos con tareas de propósito general.

Posteriormente, como respuesta a la necesidad de integrar nuevos dispositivos con arquitecturas de propósito específico con procesadores de carácter general, apareció OpenCL. OpenCL es un estándar para computación en entornos heterogéneos: *GPUs* (u otros dispositivos aceleradores) y procesadores de carácter general. No obstante, la utilización de una *API* para expresar la ejecución de tareas (*kernels*) heterogéneas supone una carga adicional para el desarrollador ya que es necesario un gran numero de operaciones y en un orden específico. Todo este proceso es muy propenso a errores.

1.1. Objetivos

El objetivo principal del proyecto es facilitar la tarea del programador mediante el desarrollo de una serie de herramientas que agilicen la adaptación o desarrollo de aplicaciones para ser ejecutadas en entornos heterogéneos (*GPU* y *CPU*) a través de la *API* OpenCL.

Para conseguir este objetivo extenderemos las funcionalidades del modelo de programación paralela OmpSs (extensión del modelo OpenMP 3.0) y crearemos una herramienta para transformar el código *C* con directivas OmpSs a código con tareas OmpSs que ejecutarán código OpenCL.

Estos cambios en el modelo facilitará el desarrollo de aplicaciones para sistemas heterogéneos y la ejecución eficiente de tareas en paralelo.

Por un lado, se desarrollará un compilador fuente a fuente que, a través de una serie de directivas basadas en OmpSs (OpenMP extendido), proporcionadas por el usuario,

sea capaz de generar todo el código necesario para poder hacer uso de *GPUs* y *CPUs* del sistema.

Por otro lado, se desarrollará también una librería o *runtime* que permitirá, en tiempo de ejecución, la correcta inicialización de los dispositivos y estructuras de datos, y la gestión de todos los recursos computacionales (*GPUs* o *CPUs*). Este *runtime* realizará la planificación de las tareas a ejecutar y la gestión del paso y comprobación de parámetros con tal de mejorar la localidad de los datos. Además, éste estará preparado para obtener datos sobre el rendimiento, facilitando el análisis de las aplicaciones a través de un sistema de trazas basado en Paraver, un entorno de visualización y análisis de trazas desarrollado por el BSC[1].

Finalmente se procederá a la adaptación de un conjunto de aplicaciones pertenecientes a un *benchmark* (aplicaciones de pruebas de rendimiento) de computación heterogénea. En concreto, estas aplicaciones son un subconjunto del Rodinia Benchmark [3], desarrollado por la Universidad de Virginia (EEUU). Los objetivos de la adaptación de estas aplicaciones son: validar el correcto funcionamiento de nuestras herramientas, analizar el rendimiento de los programas generados, y localizar cuellos de botella en la ejecución de los programas con tal de mejorar las herramientas desarrolladas.

1.1.1. Objetivos específicos

- Desarrollo del compilador.
    - Dar soporte a las cláusulas y construcciones:
        - Construcción *task*: Obtener información sobre los parámetros y dependencias de datos.
        - Construcción *for* y *parallel for*: Obtener información sobre el espacio de

[1] Barcelona Supercomputing Center

iteraciones del bucle, que pasaremos a ejecutar en paralelo en un dispositivo.

        - Cláusula *present*: Añadir una nueva cláusula al modelo de programación paralela OmpSs.
        - Construcción *flush*: Añadir soporte para el trabajo con dispositivos con memoria propia.
    - Traducción de los fragmentos a acelerar a código *kernel* de OpenCL.
    - Generar el código del *host* para la utilización del *runtime*.
- Desarrollo del *runtime*:
    - Gestión de recursos de OpenCL.
        - Inicializar y gestionar los dispositivos.
        - Inicializar y gestionar los objetos de memoria.
        - Inicializar estructuras de datos necesarias para la ejecución (gestión del paso de parámetros, etc).
        - Optimizaciones en base a los datos obtenidos con el análisis.
    - Planificación de tareas.
- Verificación del correcto funcionamiento de las herramientas.
- Adaptación de un subconjunto de aplicaciones del Rodinia Benchmark.
- Análisis de rendimiento.
- Redacción de la memoria del proyecto.

1.2. Metodología

A continuación explicamos la metodología general que se ha usado para alcanzar el objetivo principal fijado.

En primer lugar se procede al desarrollo de una aplicación simple (suma de vectores) con el fin de utilizar la *API* de OpenCL y aprender así sobre el uso de ésta. Esta primera aplicación también nos permitió, debido a su sencillez, efectuar las primeras pruebas de nuestras herramientas.

El siguiente paso es el desarrolló de una aplicación más compleja que se ejecutó de forma paralela Con ello se quiso determinar determinar cuáles eran los requisitos para ejecutar tareas en paralelo.

En este caso, también usamos la aplicación para obtener datos sobre el rendimiento así como realizar verificaciones de nuestras herramientas.

Una vez determinados los requisitos necesarios para ejecutar tareas en paralelo con OpenCL, pasamos a determinar las construcciones y/o cláusulas de OmpSs que ayudasen a expresar la ejecución en paralelo de una tarea y así poder hacer la transformación de código adecuada.

Una vez determinadas las cláusulas y/o construcciones. se procedió a escribir el compilador fuente a fuente para dar soporte a estas construcciones, haciendo uso de las herramientas proporcionadas por la infraestructura de compilación usada (Mercurium).

Después de haber añadido las fases de compilación necesarias a Mercurium, se implementaron las funcionalidades necesarias en el *runtime* para la ejecución de tareas en paralelo usando la *API* de OpenCL. También se añadió un sistema de generación trazas basado en Paraver con el fin de poder analizar el rendimiento de las aplicaciones.

Una vez implementadas las funcionalidades, se procedió a realizar un análisis sobre el rendimiento del sistema para alguna aplicaciones. Tras este análisis, se añadieron nuevas funcionalidades en el compilador y el *runtime*.

La figura 1.1 muestra gráficamente el proceso de desarrollo del proyecto.

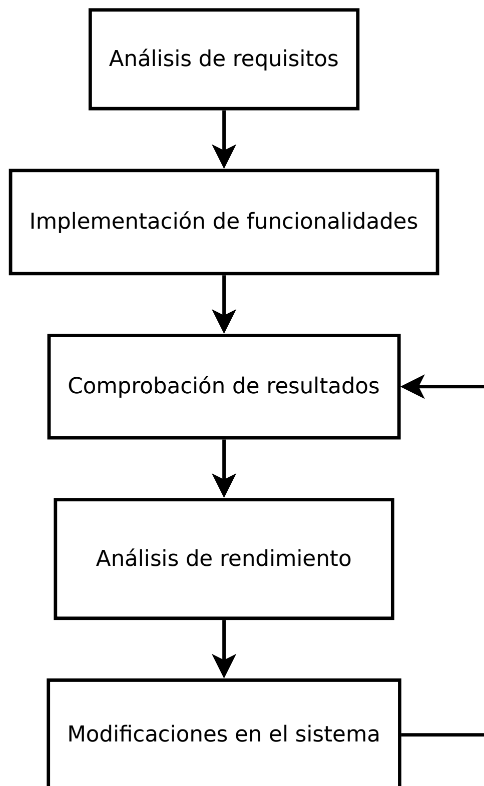

Figura 1.1: Proceso de desarrollo.

1.3. Organización de la memoria

Los capítulos 2, 3, 4 y 5 presentan las herramientas que se han utilizado a lo largo del desarrollo del proyecto. En concreto, se explican OpenMP, OmpSs, OpenCL, y Mercurium respectivamente.

A continuación, en el capítulo 6, se explica el diseño y el funcionamiento del sistema OmpSsCL. Los detalles de la implementación y fases del funcionamiento de OmpSsCL se explican en el capítulo 7. En el capítulo 8 se detalla la instalación y uso de OmpSsCL.

Los resultados y el análisis del rendimiento de las aplicaciones usando OmpSsCL se presentan en el capítulo 9. Finalmente, la planificación y costes del proyecto, y las conclusiones del proyecto se concretan en los capítulos 10 y 11, respectivamente.

Capítulo 2

OpenMP

OpenMP [1] es una *API* orientada a la programación paralela en entornos de memoria compartida. OpenMP proporciona al programador una interfaz sencilla basada en un conjunto de directivas del compilador y variables de entorno que permiten definir el comportamiento de la aplicación.

OpenMP se basa en el modelo *fork join*. En este modelo, el programa se ejecuta utilizando un solo hilo de ejecución. Cuando el programa alcanza una sección paralela, crea una serie de hilos para ejecutar la sección en paralelo (*fork*), compartiendo entre ellos la carga de trabajo. Una vez finalizada la sección, todos los hilos se sincronizan, de manera que la ejecución del programa principal sólo continuará cuando todos los hilos hayan finalizado la ejecución (*join*). En la figura 2.1 se muestra un programa ejecutando varias secciones en paralelo usando el modelo *fork-join*.

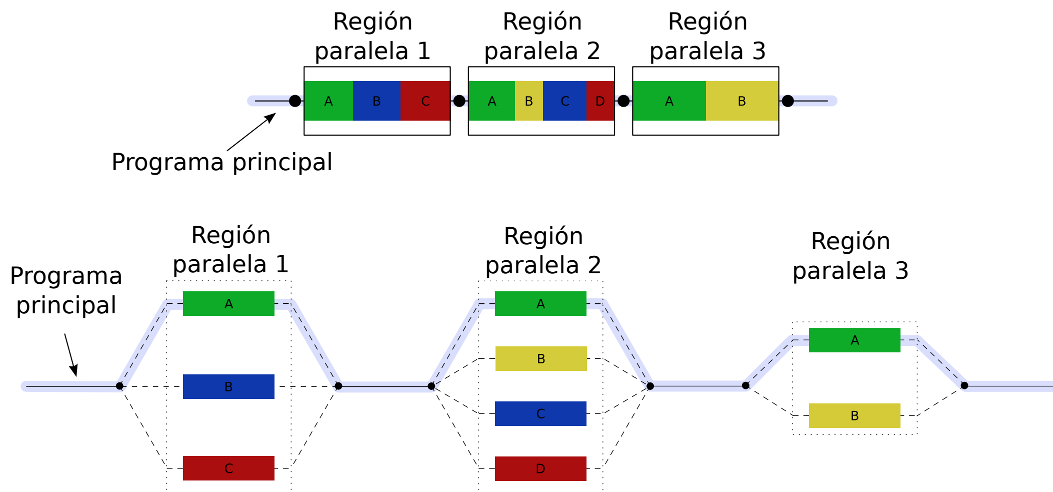

Figura 2.1: Modelo fork-join.

En el código 2.1 se muestra un ejemplo de utilización de las directivas de OpenMP para el programa de suma de vectores. En el ejemplo se está expresando con la directiva `#pragma omp parallel for` en la línea 4 del código 2.1 que se quiere crear una región paralela (directiva `parallel`) donde varios threads trabajarán repartiéndose las iteraciones del bucle for (directiva `for`).

```
1 int a[N], b[N], c[N];
2 ...
3 int i;
4 #pragma omp parallel for
5 for (i=0; i<N; i++){
6     c[i] = a[i] + b[i];
7 }
```

Código 2.1: Ejemplo de programa OpenMP.

En nuestro proyecto, usaremos un subconjunto de construcciones y cláusulas de Open-

MP detalladas a continuación:

parallel: Indica la creación de una región paralela, tal y como hemos comentado antes, habiendo una sincronización después de la región con un *join* de los *threads* creados.

for: Reparte el espacio de iteraciones del bucle for entre los diferentes threads que ejecutan ese bucle for. Del mismo modo que la directiva **parallel**, al final del bucle se efectúa una sincronización de forma implícita.

parallel for: Combinación de las dos construcciones anteriores.

nowait: Esta cláusula, colocada después de las directivas **for** o **parallel for** indica que no se realice la sincronización implícita al final de la región.

flush(*lista de variables*): Indica que se debe hacer visible el valor de las variables compartidas que aparecen en la lista a todos los threads de la región paralela.

task: Define de manera explícita una tarea, de manera que distintas instancias de la tarea puedan ser ejecutadas en paralelo.

taskwait: Define un punto de sincronización en el que se espera a que finalice la ejecución de todas las tareas creadas hasta el momento.

Capítulo 3

OmpSs

OmpSs [4] es un modelo de programación basado en OpenMP que integra sobre un solo modelo de programación características de StarSs y OpenMP [2].

OmpSs añade a OpenMP la opción de especificar para cada tarea sus dependencias de datos mediante la definición de entradas y salidas. Estas entradas y salidas corresponden a los datos que una tarea determinada necesita como entrada para ejecutarse correctamente o los datos que generará como salida.

La definición de las dependencias de datos de las tareas permite que, en tiempo de ejecución (usando el *runtime* Nanos, de la implementación de OpenSs del BSC) pueda determinarse qué tareas pueden ejecutarse dependiendo de si sus datos de entrada están o no listos.

A continuación se detallan las construcciones de OmpSs utilizadas en este proyecto.

- Construcción **task**:

  Se añaden nuevas cláusulas que amplían la construcción **task** de OpenMP indicando las dependencias de datos de la tarea. En todas ellas se especifica como parámetro una lista de variables.

  input: Indica los datos de entrada para la tarea.

output: Indica los datos de salida de la tarea.

inout: Indica que los datos especificados en ella son tanto de entrada como de salida.

- Construcción **target**:

  Permite especificar aspectos referentes al dispositivo sobre el que se ejecutará la tarea.

  device: Esta cláusula de la construcción **task** permite especificar el conjunto de dispositivos sobre los que se ejecutará la tarea.

A modo de ejemplo de utilización, el código 3.1 implementa una suma de vectores mediante la función `add` definida como tarea con dependencias de datos, especificadas en la construcción `task` a través de las cláusulas `input` y `output` en la línea 2. En este caso las dependencias de entrada son un elemento de los vectores `a` y `b` y la dependencia de salida es también un elemento del vector `c`. La principal diferencia con respecto a la versión OpenMP es que en este caso definimos explícitamente las dependencias de datos y que la tarea se ejecutará en la *CPU* (cláusula `device(smp)`).

```
1 #pragma omp target device(smp)
2 #pragma omp task input(a[1], b[1]) output(c[1])
3 void add(int *a, int *b, int *c){
4     *c = *a + *b;
5 }
6 ...
7 for (i=0; i<N; i++){
8   add(&a[i], &b[i], &c[i]);
9 }
```

Código 3.1: Ejemplo de programa OmpSs.

3.1. Nanos

Nanos es un entorno de ejecución que proporciona una infraestructura flexible, orientada al desarrollo e investigación de modelos de programación paralelos basados en arquitecturas multiprocesador de memoria compartida.

Nanos++, soportado por la infraestructura de compilación Mercurium (ver capítulo 5), es el runtime de la implementación de OpenMP y OmpSs del BSC. Nanos++ proporciona el soporte necesario para la gestión y planificación de la ejecución en paralelo de tareas.

Otra de las funcionalidades que proporciona el runtime de Nanos es el soporte para instrumentación y obtención de trazas de las ejecuciones. Estas trazas son compatibles con la herramienta de análisis Paraver.

En este proyecto hemos aprovechado las funcionalidades existentes de Nanos para la planificación y gestión de tareas, a su vez que hemos extendido la instrumentación a las ejecuciones en los dispositivos gestionados por OpenCL.

Capítulo 4

OpenCL

OpenCL es un estándar para la programación paralela en entornos heterogéneos.

OpenCL nace de la necesidad de unificar el acceso a dispositivos. Hasta el momento, cada fabricante proporcionaba una *API* para el acceso a sus dispositivos, existiendo así varias *APIs* distintas como *CUDA*, de NVIDIA o *StreamSDK* de AMD/ATi. Esto implica que el programador debe desarrollar cada aplicación para cada una de las plataformas existentes para poder dar soporte a los distintos dispositivos.

La figura 4.1 muestra el diagrama de capas de OpenCL. La aplicación accede al *framework* de OpenCL mediante una *API* y a través de los *kernels*. En tiempo de ejecución se accede a las funcionalidades de los dispositivos usando un *runtime* que accederá a los drivers de los dispositivos proporcionados por el fabricante. Aunque el driver y el *runtime* son partes independientes, es común que ambos sean distribuidos conjuntamente por los fabricantes de dispositivos.

Los *kernels* de OpenCL están escritos en una extensión del estándar C99 que añade algunas funcionalidades para dar soporte al paralelismo en las aplicaciones (funciones de sincronización, etc.), funciones matemáticas y otras utilidades. Estos programas deberán compilarse (típicamente en tiempo de ejecución) para el dispositivo concreto sobre el que se vayan a ejecutar.

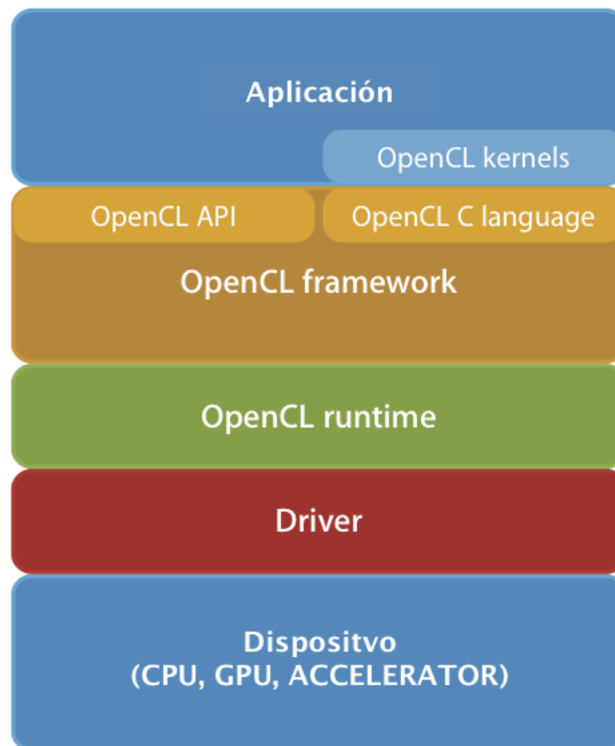

Figura 4.1: Diagrama de capas de OpenCL.

4.1. API OpenCL

A continuación se describen los componentes principales de la API de OpenCL [5]:

Platform : Representa la implementación del *runtime* de OpenCL, típicamente implementada por el fabricante de hardware. A partir de la plataforma se crean los objetos necesarios para hacer funcionar los dispositivos asociados a esta plataforma. Una plataforma puede tener varios dispositivos y contextos de ejecución.

DeviceID : Representa un dispositivo. Permite crear el contexto de ejecución de los programas y las colas de trabajo para el dispositivo.

Context : Representa el contexto de ejecución de los programas, es decir, todos los elementos necesarios para la ejecución del *kernel*. Concretamente uno o más dis-

positivos, el programa (o programas) a ejecutar, colas de ejecución, eventos y los objetos de memoria que usará el programa en su ejecución.

Program : Representa un programa OpenCL, bien sea en su versión como código fuente, código intermedio o código maquina de un dispositivo concreto dependiendo de la implementación. Un objeto **program** está asociado a un contexto (**context**), a algún dispositivo (**DeviceID**) y puede tener asociados varios *kernels*.

Kernel : Contiene una sola función compilada del objeto **program**. Ésta será la tarea que se envíe a ejecutar en el dispositivo.

CommandQueue : Es una cola de trabajo de un dispositivo. Una cola contiene comandos como copias de memoria o la ejecución de programas en el dispositivo. Cada cola está asociada a un único contexto y dispositivo.

MemObject : Representa un bloque de memoria dentro de un contexto y permite la copia de datos con los dispositivos más allá de los tipos simples. Pueden ser del tipo *Buffer* (buffer de propósito general) o *Image* (orientado a imágenes). Este último no se utilizará en el proyecto puesto que no trataremos las imágenes de manera específica, así como tampoco sol objetos del tipo **Sampler**, utilizados únicamente para el proceso de imágenes.

Event : Representa un evento dentro de una cola (CommandQueue), permite la sincronización con algunas operaciones asíncronas de OpenCL como las copias de datos o la ejecución de tareas realizadas dentro del mismo contacto.

A continuación se describen brevemente algunas de las funciones que permiten hacer un uso básico de OpenCL:

clGetPlatformIDs(): Obtiene todas las plataformas disponibles del sistema.

clGetDeviceIDs(): Obtiene todos los dispositivos disponibles en una plataforma.

clCreateContext(): Crea un contexto para poder gestionar colas de comandos (*command_queue*), objetos de memoria (*buffers*) y objetos de tipo programa (*program* y *kernel*).

clCreateCommandQueue(): Crea una cola de comandos para un dispositivo.

clCreateBuffer(): Crea un objeto de memoria (o *buffer*).

clCreateProgramWithSource(): Crea un programa para un contexto determinado a partir de su código fuente.

clBuildProgram(): Compila un programa.

clSetKernelArg(): Asigna un valor a un determinado argumento del *kernel*.

clEnqueueWriteBuffer() Añade a la cola de ejecución el comando para escribir un *buffer* del dispositivo.

clEnqueueReadBuffer(): Añade a la cola de ejecución el comando para leer un *buffer* del dispositivo.

clEnqueueTask(): Añade a la cola de ejecución un *kernel*, para ser ejecutado una única vez.

clEnqueueNDRangeKernel(): De forma similar a **clEnqueueTask**, añade a la cola de ejecución un *kernel*, pero en este caso ejecutará una serie de instancias del *kernel* en paralelo según los parámetros global_work_offset y global_work_size, que definen el espacio de iteraciones en que se ejecutará en *kernel*.

En la figura 4.2 se muestran todas las relaciones entre los distintos componentes en forma de diagrama UML.

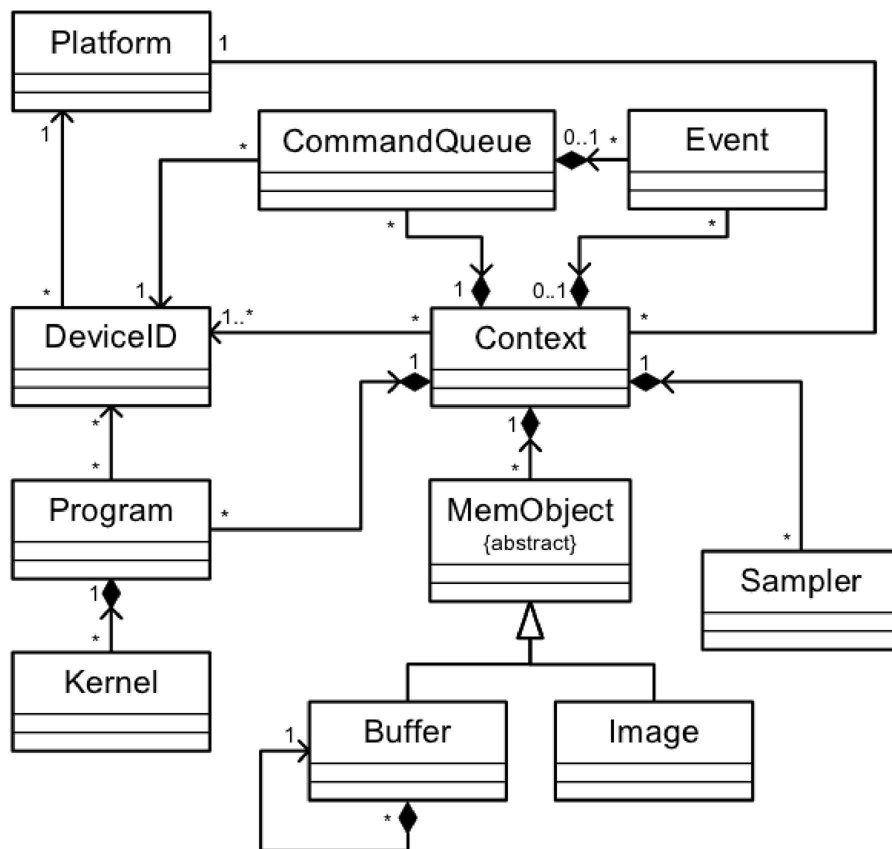

Figura 4.2: Modelo de datos de OpenCL.

El código 4.1 muestra el ejemplo de una suma de vectores implementado con OpenCL. En el ejemplo se pueden observar las numerosas operaciones que son necesarias para ejecutar un código en una *GPU*. Nótese que hemos eliminado todo el control de errores para que el código no fuera excesivamente largo. A continuación enumeramos las operaciones que se realizan en el código 4.1.

1. Obtener una plataforma del sistema y un dispositivo de tipo *GPU* (líneas 16 y 17).

2. Crear el contexto de ejecución (línea 19).

3. Crear la cola de tareas para el dispositivo (línea 20).

4. Crear los objetos de memoria de OpenCL. Se crean dos buffers que serán leídos por

el *kernel*. (líneas 23 y 24) y uno, donde se guardará el resultado, que será escrito por el *kernel* (línea 25).

5. Crear y compilar el *kernel* (líneas 29 a 32).
6. Paso de parámetros al *kernel* (líneas 35 a 37).
7. Copia de datos al dispositivo (líneas 40 y 41).
8. Ejecución del *kernel* (línea 43).
9. Lectura de resultados del dispositivo (línea 45).

El código 4.2 corresponde al *kernel* que se compila en el paso 5 y será ejecutado en el dispositivo, en este caso una *GPU*. Cabe destacar el uso de elementos propios del lenguaje OpenCL:

__kernel: Indica que la función será ejecutada sobre un dispositivo.

__global: Indica que el puntero apunta a memoria global del dispositivo y por lo tanto los datos son accesibles por todas las instancias del *kernel*.

get_global_id(0): Retorna el identificados global de la instancia de ejecución del *kernel*.

```
1 char* loadSource(const char* path, int *size){
2     ...
3 }
4
5 cl_context context;
6 cl_command_queue commandQueue;
7 cl_platform_id cpPlatform;
8 cl_device_id device;
```

```
9  cl_program cprogram;
10 cl_kernel kernel;
11 cl_program cprogram;
12
13 cl_int err;
14 cl_uint numPlatforms, numDevs;
15 //opbtener plataformas y dispositivos
16 clGetPlatformIDs(1, &cpPlatform, &numPlatforms);
17 clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device, &
     numDevs);
18 //crear contexto
19 context = clCreateContext(NULL, numDevs, &device, NULL, NULL, &
     err);
20 commandQueue = clCreateCommandQueue(context, device,
     CL_QUEUE_PROFILING_ENABLE, &err);
21
22 cl_mem cl_a, cl_b, cl_c;
23 cl_a = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int)*
     size, NULL, &err);
24 cl_b = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int)*
     size, NULL, &err);
25 cl_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int)*
     size, NULL, &err);
26
27 //crear kernel
28 int srcLen;
29 char* source = loadSource("vectorAdd.cl", &srcLen);
```

```
cprogram = clCreateProgramWithSource(context, 1, (const char**)&
    source, 0, &err);
clBuildProgram(cprogram, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(cprogram, "vectorAdd", &err);

//paso de parametros al kernel
clSetKernelArg(kernel, 0, sizeof(cl_mem), &cl_a);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &cl_b);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &cl_c);

//copia de buffers al dispositivo
clEnqueueWriteBuffer(commandQueue, cl_a, CL_FALSE, 0, sizeof(int
    )*size, a, 0, NULL, &ev_ba);
clEnqueueWriteBuffer(commandQueue, cl_b, CL_FALSE, 0, sizeof(int
    )*size, b, 0, NULL, &ev_bb);
//ejecucion del kernel (poner en cola)
clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL, &size,
    NULL, 0, NULL, &ev_k);
//lectura de resultados
clEnqueueReadBuffer(commandQueue, cl_c, CL_TRUE, 0, size*sizeof(
    int), c, 0, NULL, &ev_bc);
...
```

Código 4.1: Código del *host* de la suma de vectores implementada con OpenCL.

```
__kernel void vectorAdd(__global const int *a, __global const
    int *b, __global int *c)
```

```
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

Código 4.2: Código del *kernel* OpenCL de la suma de vectores.

Como hemos visto en los códigos 4.1 y 4.2, para ejecutar una tarea simple como es una suma de vectores es necesario efectuar un gran número de operaciones. Como consecuencia, el resultado un código largo y de difícil lectura.

Capítulo 5

Mercurium

Mercurium es un compilador *fuente a fuente* que permite de manera rápida y simple efectuar transformaciones de alto nivel en el código fuente de entrada. Actualmente Mercurium soporta, como lenguajes de entrada, *C*, *C++* y *Fortran*. Cabe destacar también que Mercurium no generará código binario como salida. Para ello es necesario otra herramienta para realizar esta tarea, como por ejemplo, el compilador gcc de GNU.

La compilación de un fichero a través de Mercurium es un proceso que consta de varias etapas, como se puede ver en la figura 5.1. Primero se realiza un análisis léxico del código de entrada en el módulo analizador *C*/*C++* del compilador. En esta etapa se genera, además, una representación intermedia del código con la que se operará durante todo el proceso de compilación.

La siguiente etapa es el proceso de compilación basado por fases. Las fases se encadenan en forma de filtros o *pipeline*, de tal forma que la salida de una fase se convierte en la entrada de la siguiente. Es en esta etapa donde añadiremos todas las operaciones necesarias para obtener el código que permita el funcionamiento de nuestras herramientas, añadiendo una nueva fase a la cadena.

La etapa final del compilador es la etapa de *prettyprinting*, donde a partir de la representación intermedia, se genera un fichero de código fuente en un lenguaje determinado.

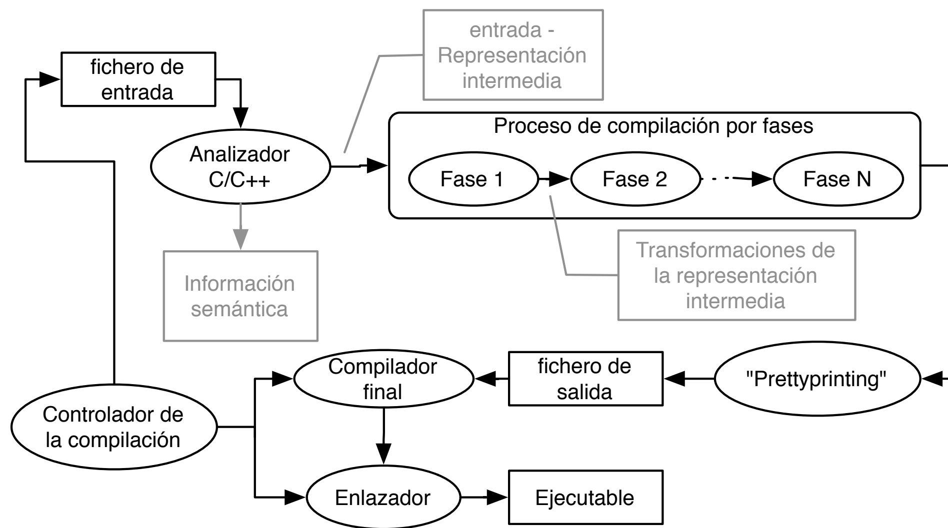

Figura 5.1: Diagrama de fases de Mercurium.

Una vez generado el fichero fuente, Mercurium invoca una serie de herramientas externas (como gcc) con el objetivo de compilar y enlazar el código fuente que se ha generado. De esta manera al final del proceso obtenemos un programa ejecutable.

Capítulo 6

Diseño y funcionamiento

OmpSsCL es un entorno de compilación y ejecución que, a través del modelo de programación paralela OmpSs nos permite ofrecer funcionalidades de OpenCL de forma sencilla y eficiente.

Nuestro entorno está dividido en dos partes bien diferenciadas:

1. Un compilador fuente a fuente.

2. Una librería de ejecución que abstrae la funcionalidad de OpenCL.

Usando este entorno OmpSsCL, el programador podrá acelerar con OpenCL, en una *GPU* o un procesador *multicore*, las funciones etiquetadas con la construcción `task` que sean cuerpo de un bucle `for` cuyas iteraciones se puedan ejecutar en paralelo (etiquetado con las directivas `omp for` o `omp parallel for`.

Las directivas (construcciones y cláusulas) a las que damos soporte son las siguientes:

- Construcción **target**:

  device(*lista de dispositivos*) : Determina en que dispositivo se ejecutará la tarea, cpu (o smp) y gpu.

present(*lista de variables*) : Los parámetros que aparecen en la lista no deben eliminarse del dispositivo una vez finalizada la tarea. En el caso de un parámetro de entrada también indica que si el parámetro ya se había copiado al dispositivo, éste no se debe volver a copiar. En el caso de un parámetro de salida, el dato no se copiará automáticamente a la memoria del *host* al finalizar la tarea.

- Construcción **task**:

  Las siguientes cláusulas de esta construcción definen tanto las dependencias de la tarea como las transferencias de datos que se deben realizar con el dispositivo. En estas cláusulas se especifica una lista de argumentos. Los argumentos se especifican como vectores o partes de un vector. Por ejemplo, un parámetro del tipo `v[N]` indica que la función utilizará las `N` primeras posiciones del vector `v` y `v[B:E]` indica que la tarea usar los datos de v entre las posiciones B y E.

  input(*lista de argumentos*): Indica cuales de los argumentos son de entrada.

  output(*lista de argumentos*): Indica cuales de los argumentos son de salida.

  inout(*lista de argumentos*): Los argumentos presentes en esta cláusula serán considerados de entrada y salida al mismo tiempo.

- Construcción **omp for** y **omp parallel for**.

  Esta construcción indica que las iteraciones en el bucle se pueden ejecutar en paralelo, y por lo tanto no tienen dependencias entre ellas. Por otra parte, cuando esta construcción aparece en un bucle cuyo cuerpo sea una tarea, ésta podrá ser ejecutada en paralelo en un dispositivo.

  nowait : Cláusula que especifica que los *threads* que acaban el bucle paralelo `for` no tienen que esperarse al resto de threads. En el caso concreto de OmpSsCL también indica que el *host* puede continuar ejecutando el programa principal sin esperar a que acabe la ejecución de la tarea en el dispositivo.

- **flush (*lista de variables*)**:

  Con esta construcción se garantiza que al terminar la ejecución del bucle, los argumentos presentes en esta cláusula serán visibles al *host*. Esta cláusula se debe especificar dentro del bucle, y no dentro de la construcción `omp (parallel) for`

La figura 6.1 muestra el funcionamiento del sistema. El compilador fuente a fuente Mercurium recibe como entrada un programa escrito en *C* con una serie de directivas que especifican las tareas que se transformarán, indicando sus dependencias y el dispositivo en que se ejecutarán.

El primer paso es el *parser*, donde el compilador fuente a fuente genera una representación intermedia con las que se trabaja durante todo el proceso de transformaciones.

Una vez que se ha obtenido la representación intermedia, pasamos a las fases de transformación del código (fases de compilación en la figura 6.1), que podemos dividir principalmente en tres: OpenMP, OmpSs to OpenCL y Nanos.

En la primera de las fases de compilación se obtiene información general sobre las tareas: dependencias de entrada y/o salida y el dispositivo sobre el que deben ejecutarse.

En la siguiente fase (OmpSs to OpenCL) se genera el *kernel* OpenCL que implementa las tareas y que serán utilizados por el *runtime* OpenCL en tiempo de ejecución. En esta fase también se transforman las tareas del código de entrada en nuevas tareas cuya única funcionalidad es la de invocar a los *kernels* de OpenCL mediante el *runtime* de OmpSsCL. Además, a estas nuevas tareas se las etiqueta como tareas para que en la fase posterior del proceso de transformación de código (Nanos) se generen las llamadas al *runtime* OmpSs (Nanos++); con tal de que el *runtime* OmpSs se encargue del control de dependencias en tiempo de ejecución.

Como resultado de estas fases hemos obtenido un código *C* del *host* donde hay nuevas tareas que invocan, mediante el *runtime* de OmpSsCL, a los códigos de *kernel* generados durante las fases de compilación a partir de las tareas especificadas en el código original. Las nuevas tareas se crean utilizando el *runtime* de OmpSs, Nanos++.

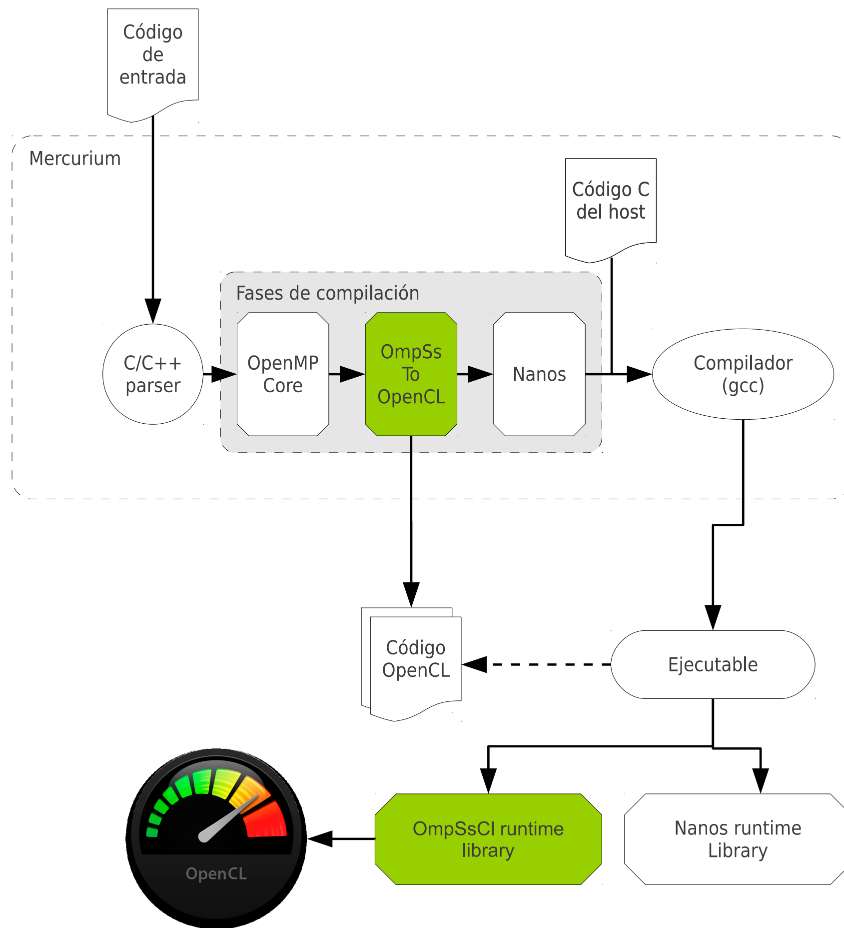

Figura 6.1: Diagrama del funcionamiento de OmpSsCL.

Una vez finalizadas todas estas fases, el código del *host* se compila con un compilador externo , como por ejemplo gcc, para generar un binario ejecutable.

La figura 6.2 muestra el diagrama de capas de OmpSsCL, para una aplicación escrita en OmpSsCL. El código *host* resultante de todas las transformaciones hace uso del *runtime* de Nanos y del *runtime* OmpSsCL. En el caso del *runtime* de Nanos, éste se usa para llevar a cabo toda la gestión de dependencias de las tareas (tareas que invocan los *kernels* generados) y la instrumentación de la ejecución de éstas. Para el caso del *runtime* de OmpSsCL, éste se utiliza para gestionar la invocación de *kernels* OpenCL y los recursos de OpenCL, abstrayendo la *API* de OpenCL al usuario.

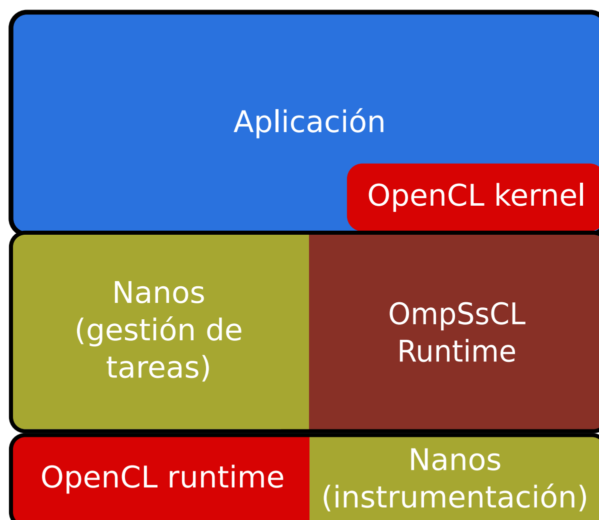

Figura 6.2: Diagrama de capas de una aplicación OmpSsCL.

En el capítulo 7 se describe en detalle el contenido de todo el código generado y el proceso de generación de código.

En el código 6.1 se muestra un ejemplo sencillo de código de entrada, que implementa la suma de dos vectores. En las líneas 4 y 5 del código 6.1 se han añadido dos pragmas para definir la tarea y el dispositivo a ejecutar. En la línea 4 se especifica que la tarea debe ejecutarse en un dispositivo de tipo *GPU* (`target device(gpu)`). En la línea 5 se especifica que la función es una tarea con parámetros de entrada `a` y `b` y de salida `c`. Los parámetros de la tarea son todos de un único elemento.

En la línea 10 se ha añadido la construcción `parallel for` para indicar que las distintas iteraciones del bucle pueden ejecutarse en paralelo. También se especifica, de manera indirecta, a través del bucle `for`, cual es el espacio de iteraciones y cuales serán los datos necesarios para que la función opere correctamente.

```
1 int a[VS], b[VS], c[VS];
2 int i;
```

```
3  ...
4  #pragma omp target device(gpu)
5  #pragma omp task input(a[1], b[1]) output(c[1])
6  void add(int *a, int *b, int *c){
7       *c = *a + *b;
8  }
9  ...
10 #pragma omp parallel for
11 for (i=0; i<VS; i++){
12      add(&a[i],&b[i],&c[i]);
13 }
14 ...
```

Código 6.1: Ejemplo de código de entrada escrito para OmpSsCL.

Capítulo 7

Implementación

En este capítulo se detalla el proceso de compilación y de transformación de código efectuado por nuestro sistema, la *API* de nuestro *runtime* y la gestión y optimización de recursos de OpenCL.

7.1. Transformación de código

En este apartado se explican las transformaciones de código fuente a código fuente realizadas por OmpssCL. En particular, se explica la traducción de un código en *C* con directivas OmpSs a código OpenCL (*kernels*) y código del *host* con llamadas a las librerías de soporte a la ejecución (*API* del *runtime*).

A continuación explicamos cómo transformar el código escrito por el programador en *C* con directivas OmpSs a:

1. Código con llamadas a la *API* de nuestro *runtime*, que gestiona el uso de OpenCL.

2. Kernels OpenCL.

El proceso de análisis y transformación de código que hemos desarrollado a lo largo de este proyecto se realiza después de la fase inicial de Mercurium. Este proceso inicial, que incluye la fase de análisis de directivas OmpSs, realiza un análisis del código para obtener

una representación intermedia del código e información de las tareas como su entrada y salida.

Nuestra transformación se puede dividir en varias partes: la generación del código de inicialización del runtime y las tareas, la generación del código de ejecución de las tareas y la generación de código OpenCL.

A continuación mostramos un ejemplo de una suma de dos vectores en el código 7.1, a partir del cual se explicarán todas las partes de la generación de código.

```
#pragma omp target device(gpu)
#pragma omp task input(a[1], b[1]) output(c[1])
void addPar(int *a, int *b, int *c){
    *c = *a + *b;
}

int main(int argc, char *argv[]){
...
    int i;
    #pragma omp parallel for
    for (i=0; i<VS; i++){
        addPar(&a[i],&b[i],&c[i]);
    }
...
}
```

Código 7.1: Ejemplo de código de entrada para la suma de dos vectores para OmpSsCL.

El código 7.1 muestra un bucle `for` en la línea 11 que, según indica el programador

con la construcción `omp parallel for`, se puede paralelizar. Dentro del bucle hay una llamada a la función `addPar`, que está definida como `task` en la línea 2 del código 7.1. Además se indica que tiene como parámetros de entradas `a` y `b` y como salida `c`. La tarea `addPar` se ejecutará en un dispositivo de tipo *GPU*, según se especifica en la línea 1 del código 7.1 mediante la construcción `target device(gpu)`.

7.1.1. Generación de código OpenCL

Por un lado el compilador genera el código OpenCL de los *kernels* correspondientes a cada tarea especificada por el programador. Para ello se analizan los pragmas especificados por el programador con tal de obtener información de la dirección de los parámetros (entrada, salida o entrada/salida), el tipo y tamaño de éstos, el acceso que se realiza sobre ellos (desplazamiento respecto a la dirección base) y si son accedidos o no por la variable de inducción del bucle.
El código 7.2 muestra como se ha transformado la tarea `addPar` definida en el código 7.1.

En la generación de código de OpenCL se añade a la cabecera de la función (*kernel*) todas las directivas de OpenCL necesarias, tales como `__kernel`, para indicar que se está definiendo un *kernel* de OpenCL o `__global`, para indicar que los argumentos de tipo vector son visibles para todas las instancias del *kernel* (y se almacenan en la memoria global del dispositivo). También se añade una serie de parámetros extra correspondientes al desplazamiento de los argumentos para acceder a la posición correcta de cada vector en cada ejecución del *kernel*.

```
1     __kernel void addPar(__global int *a, __global int *b,
          __global int *c, int __cl_offset_arg_a, int
          __cl_offset_arg_b, int __cl_offset_arg_c){
2 int __cl_local_i = get_global_id(0);
3 c[__cl_offset_arg_c + __cl_local_i] = a[__cl_offset_arg_a +
```

```
    __cl_local_i] + b[__cl_offset_arg_b + __cl_local_i];
}
```

Código 7.2: Código OpenCL generado para implementar el kernel de la suma de vectores.

El código 7.2 muestra como se ha transformado la tarea `addPar` a *kernel* OpenCL.

En el cuerpo de la función se declara una nueva variable de tipo entero inicializada con el valor de `get_global_id(0)` (línea 2 del código 7.2). Esto hará que en cada instancia de ejecución del *kernel*, la variable tome el valor del identificador global de la instancia. A lo largo de la ejecución, esta expresión de OpenCL devolverá todos los valores del espacio de iteraciones del bucle de la línea 11 del código 7.1. Es decir, su función es equivalente a la variable de inducción del bucle.

Para acceder correctamente a los datos de las variables accedidas por el *kernel*, se añade un desplazamiento propio de cada vector. Esto se hace para compensar los posibles desplazamientos de los argumentos de entrada con respecto a la variable de inducción. En el código 7.2 los desplazamientos están dentro de las variables `__cl_offset_arg_X`. En este caso concreto, valdrían todos 0, ya que no hay desplazamiento alguno en ninguno de los argumentos en el código 7.1.

Por ejemplo, en el código 7.3 se define una tarea que utiliza, como entrada los elementos entre `B` y `E` del vector `v` (cláusula `input(v[B:E])` en la línea 2) y sólo el primer elemento `o` (cláusula `output(o[1])` en la línea 2). En este caso, Puesto que sólo se utilizan los elementos entre `B` y `E`, el desplazamiento corresponde a `-B`.

```
#pragma omp target device(gpu)
#pragma omp task input(v[B:E]) output(o[1])
void foo(int* v, int*o){
    int i;
    for(i=B; i<E; i++){
```

```
6            *o += v[i];
7        }
8  }
9  ...
10 #pragma omp parallel for
11 for (i=0; i<N; i++){
12       foo(&v[i], &o[i]);
13 }
14 ...
```

Código 7.3: Ejemplo de código con un parámetro con un rango de valores `[B:E]`.

En el código 7.4 se muestra el código OpenCL generado que implementa la tarea `foo` del código 7.3. La variable de desplazamiento para el argumento `v` (`__cl_offset_arg_v`), valdría `-B` puesto que es el desplazamiento que se ha definido para este parámetro en la línea 2 del código 7.3 es `B` (`v[B:E]`). En la primera ejecución, la llamada a `get_global_id(0)` retorna 0, y es necesario acceder a la primera posición del vector `v` que se ha copiado al dispositivo (en la sección 7.3 se muestra detalladamente como se efectúan las copias de datos).
De este modo, en la línea 7 se suma el desplazamiento en los accesos al vector `v`. Por otro lado, `__cl_offset_arg_o` vale 0 puesto que en la tarea se define que se usa la primera posición del vector.

```
1 __kernel void foo(__global int *v, __global int *o, int
      __cl_offset_arg_v , int __cl_offset_arg_o){
2 int __cl_local_i = get_global_id(0);
3 int i;
4     //B y E se expanden al contenido de la macro
```

```
5       for (i = B; i < E; i++)
6       {
7           o[__cl_offset_arg_o + __cl_local_i] += v[(i) +
                __cl_offset_arg_v + __cl_local_i];
8       }
9   }
```

Código 7.4: Ejemplo de código OpenCL con desplazamientos.

7.1.2. Generación de código del host

La primera transformación que se efectúa en el código del *host* es insertar el código necesario para inicializar el *runtime* y las tareas.

Este código se inserta al principio de la función principal del programa (`main`). La inicialización del *runtime* y la obtención de los dispositivos y plataformas del sistema se realiza en el momento de crear la primera tarea. De esta forma, si no hay tareas, tampoco se invierte tiempo en la inicialización del *runtime*.

El código 7.5 muestra la transformación del código 7.1 de ejemplo para la parte del *host*. En este código, la inicialización del *runtime* y las tareas se realiza en las líneas 5 a 12 del código 7.5. Concretamente, en la línea 5 del código se inicializa y compila el *kernel* que implementa la tarea `addPar`. Para ello es necesario especificar el fichero que contiene el *kernel* y su nombre (la *API* del *runtime* se especifica con detalle en la sección 7.2). En la creación de la primera tarea (`addPar`) también se inicializa el *runtime* y se obtienen todos los dispositivos y plataformas OpenCL del sistema.

Justo después se definen los parámetros del *kernel* `addPar` mediante las llamadas a `clnAddArgToTask` (líneas 6 a 12 del código 7.5). Para cada tarea se deben definir sus parámetros llamando a la *API* de nuestro *runtime*, en estas llamadas se especifica el dispositivo, para que tarea son y la dirección del parámetro (entrada/salida). Estas

llamadas (tantas como argumentos se definan en la tarea) se deben efectuar en el mismo orden en que se encuentran los parámetros en la definición de la tarea. Como se observa en el código 7.5, se efectúan tantas llamadas como argumentos hay definidos en el *kernel* `addPar` del código 7.2 (parámetros definidos en la tarea más desplazamientos).

```
1  ...
2  int main(int argc, char *argv[])
3  {
4      clnCreateTask(CLN_DEVICE_TYPE_GPU, "vectorAdd_parTest.
           c_addPar-parallel_ssclncc.cl", "addPar", &
           __cln_global_var_taskId0);
5      clnAddArgToTask(CLN_DEVICE_TYPE_GPU,
           __cln_global_var_taskId0, CLN_ARG_IN);
6      clnAddArgToTask(CLN_DEVICE_TYPE_GPU,
           __cln_global_var_taskId0, CLN_ARG_IN);
7      clnAddArgToTask(CLN_DEVICE_TYPE_GPU,
           __cln_global_var_taskId0, CLN_ARG_OUT);
8      //offset arguments
9      clnAddArgToTask(CLN_DEVICE_TYPE_GPU,
           __cln_global_var_taskId0, CLN_ARG_IN_BASIC);
10     clnAddArgToTask(CLN_DEVICE_TYPE_GPU,
           __cln_global_var_taskId0, CLN_ARG_IN_BASIC);
11     clnAddArgToTask(CLN_DEVICE_TYPE_GPU,
           __cln_global_var_taskId0, CLN_ARG_IN_BASIC);
```

Código 7.5: Código del *host* generado automáticamente para la inicialización del entorno y las tareas expresadas en OmpSsCL.

Una vez se ha inicializado el *runtime* y las tareas, se pasará a explicar como se transforma el código de las llamadas a tareas para que éstas se ejecuten en dispositivo concreto por medio de OpenCL. En concreto, detallaremos la transformación que se realiza en la llamada a una tarea que se encuentra en el cuerpo de un bucle paralelo. Para este caso, el código se transforma en un `NDRange`[5] de OpenCL cuya tarea es un *kernel* OpenCL con el código de la tarea en cuestión.

El código 7.6 corresponde a la traducción de las líneas 10, 11 y 12 del código 7.1, incluyendo los parámetros necesarios para la ejecución del *kernel*.

Para realizar esta transformación se analiza el bucle que contiene la llamada. De este bucle se obtienen las cotas superior e inferior del bucle, lo que determinará el espacio de iteraciones del *kernel*.

En lugar del bucle, se inserta el código que efectúa el paso de parámetros, ejecuta el kernel y lee los resultados.

```
1  int *__cln_local_var_arg0 = a;
2  int *__cln_local_var_arg1 = b;
3  int *__cln_local_var_arg2 = c;
4  int __cln_local_var_arg3 = -(0) - (0);
5  int __cln_local_var_arg4 = -(0) - (0);
6  int __cln_local_var_arg5 = -(0) - (0);
7  ...
8  cln_task_instance_info __cln_local_var_taskIntanceInfo =
9  { __cln_global_var_taskId0 , CLN_DEVICE_TYPE_GPU, 6 };
10 
11 cln_task_instance_argument __cln_local_var_taskIntanceArguments
       [] = {
12 {
```

```
13    (void *)(__cln_local_var_arg0 +(0)+(0)),((VS-1)+(VS)-(0)-(0))*
         sizeof(int)
14 },{
15    (void *) (__cln_local_var_arg1 + (0)), ((KERN_LEN) - (0)) *
         sizeof(float)
16 },{
17    (void *) (__cln_local_var_arg2 + (KERN_LEN / 2) + (0)),
18    (((SIG_LEN-(KERN_LEN/2))-1)+(0)-(KERN_LEN / 2)-(0))*sizeof(
         float)
19 },{
20 (void *) &__cln_local_var_arg3, sizeof(int)
21 },{
22 (void *) &__cln_local_var_arg4, sizeof(int)
23 },{
24 (void *) &__cln_local_var_arg5, sizeof(int)
25 }
26 ...
27 };
28 size_t __cln_local_var__task_size[] = {(((VS) - 1) + 1) - (0)};
29 size_t __cln_local_var__offset[] = {0};
30 clnRunNDRangeTaskInstance(&__cln_local_var_taskIntanceInfo,
      __cln_local_var_taskIntanceArguments,
      __cln_local_var__task_size, __cln_local_var__offset);
```

Código 7.6: Código del host que invoca al *kernel* OpenCL.

En la línea 30 del código 7.6 se efectúa la llamada que ejecuta el *kernel* de la tarea. Como se puede observar también en la línea 30 del código 7.6, le debemos pasar como

parámetros la información de la tarea, los argumentos para su ejecución, el espacio de iteraciones y la cota inferior del espacio de iteraciones (inicio del bucle).

En las líneas 8 y 9 se define el primer argumento de `clnRunNDRangeTaskInstance`, que corresponde a la información de la tarea y contiene el identificador de la tarea, el dispositivo sobre el que se ejecutará la tarea y el numero de parámetros (incluyendo los desplazamientos).

Seguidamente, en las líneas de la 11 a 25 se inicializan los argumentos del *kernel*. En este vector se guarda, para cada parámetro, la dirección base de cada argumento y el tamaño de los datos. También se guardan los argumentos pertenecientes a los *offset* de los vectores.

Finalmente, en las líneas 28 y 29 del código se define el espacio de iteraciones del *kernel* en base al inicio y fin del bucle del código 7.1. En la línea 28 del código 7.6 se define el tamaño del espacio de iteraciones y en la línea 29 se define el inicio del mismo.

Estos dos últimos parámetros son vectores puesto que el *runtime* esta pensado para soportar en un futuro espacios de iteraciones de varias dimensiones. En ese caso, cada posición correspondería con una de las dimensiones del espacio de iteraciones.

Como hemos podido observar, hemos transformado el código insertando llamadas a funciones de una librería (*runtime*). Los detalles de esta librería (*API* y funcionalidades se explican en las siguientes secciones).

7.2. API del Runtime

A continuación se describen los diferentes elementos, estructuras de datos y procedimientos de la *API* de *C* nuestro runtime para la gestión de dispositivos y recursos.

7.2.1. Tipos y estructuras de datos

cln_task_id: Tipo que identifica a una tarea.

cln_device_type: Tipo enumerado que representa los distintos tipos de dispositivo, se pueden combinar varios tipos de dispositivo mediante una OR bit a bit.

cln_arg_flags: Tipo que representa el tipo de argumento (entrada, salida, etc.) pueden combinarse distintos tipos mediante una OR bit a bit.

cln_task_instance_info: Estructura de datos que contiene la información de una tarea, su identificador, tipo de dispositivo sobre el que se ejecutará y numero de argumentos.

cln_instance_argument: Estructura de datos que encapsula un puntero y su tamaño en bytes. Este tipo representa un argumento que se pasará al *kernel* cuando se ejecute.

7.2.2. Procedimientos

clnInit (cln_device_type *deviceType*): Inicializa el runtime, recibe como parámetro el tipo de dispositivos sobre los que se va a trabajar.

deviceType: Uno o más tipos de dispositivos, si hay más de un dispositivo, se deben combinar mediante OR bit a bit.

clnCreateTask (cln_device_type deviceType, char **fileName*, char **kernelName*, cln_task_id *taskId*): Crea una tarea. Recibe como parámetro el tipo de dispositivos en los que se ejecutará la tarea, la ruta al fichero que contiene el *kernel* a ejecutar y su nombre. Como parámetro de salida, devuelve el identificador de la tarea que acaba de ser creada.

fileName: Ruta el fichero que contiene el *kernel* que se está inicializando.

kernelName: Nombre del *kernel* que se va a inicializar.

taskId: (parámetro de salida) Retorna el identificador de tarea del *kernel* que se ha creado.

clnAddArgToTask (cln_device_type *deviceType*, cln_task_id *taskId*, cln_arg_flags *argFlags*): Configura un nuevo argumento que recibirá la tarea cuando se ejecute. A este procedimiento se le pasan como parámetros el tipo de dispositivo, el identificador de tarea (obtenido mediante **clnCreateTask** y los flags del argumento. Las llamadas necesarias para definir los argumentos deben ejecutarse en el mismo orden en que aparecen los argumentos en la definición de la llamada.

deviceType: Uno o más tipos de dispositivos, si hay más de un dispositivo, se pueden combinar mediante OR bit a bit.

taskId: Identificador de la tarea para la que se define el argumento.

argFlags: Flags del argumento, especifican la dirección del argumento y si debe mantenerse en el dispositivo, pueden ser combinados usando OR bit a bit.

clnRunTaskInstance (cln_task_instance_info *_taskInstanceInfo_, cln_task_instance_argument _taskInstanceArguments[]_): Ejecuta una tarea inicializada. Los parámetros que se tienen que pasar a este procedimiento son: la información de la tarea a ejecutar y un vector que representa los parámetros que se pasarán el *kernel*. Este vector debe tener un elemento por a cada uno de los parámetros que se hayan configurado para esta tarea.

taskInstanceInfo: Puntero a una estructura del tipo `cln_task_instance_info` que contiene la información de la tarea.

taskInstanceArguments: Vector de estructuras de tipo `cln_task_instance_argument` que contienen la información de los parámetros que se pasan al *kernel*. Este vector debe tener tantos elementos como argumentos se han definido para el *kernel*.

clnRunNDRangeTaskInstance (cln_task_instance_info *_taskInstanceInfo_, cln_task_instance_argument _taskInstanceArguments_[], size_t* _size_, size_t*

offset): Similar a **clnRunTaskInstance**, con la diferencia de que en este caso, en lugar de ejecutarse la tarea una sola vez, ésta se ejecuta a lo largo de un espacio de iteraciones. Además de los parámetros del procedimiento **clnRunTaskInstance**, es necesario pasar dos parámetros adicionales. Se trata del tamaño del espacio de iteraciones de la ejecución de la tarea y la cota inferior del espacio de iteraciones.

taskInstanceInfo: Puntero a una estructura del tipo `cln_task_instance_info` que contiene la información de la tarea.

taskInstanceArguments: Vector de estructuras de tipo `cln_task_instance_argument` que contienen la información de los parámetros que se pasan al kernel. Este vector debe tener tantos elementos como argumentos se han definido para el *kernel*.

size: Vector que contiene el tamaño del espacio de iteraciones, con una posición por cada dimensión del espacio. (actualmente sólo se soporta una dimensión)

offset: Vector que contiene la cota inferior del espacio de iteraciones, con un elemento por cada dimensión del mismo.

clnReportUsage(): Escribe por la salida de error estándar un pequeño informe sobre el uso de los dispositivos de OpenCL.

clnFlushDevices(): Lee todos los datos almacenados en los dispositivos y libera el espacio que éstos ocupaban en cada dispositivo.

clnFlushBuffers (cln_task_instance_argument *flushBuffers*[], int *numBuffers*): Similar a *clnFlushDevices*, pero en este caso se leen uno o más parámetro del kernel en lugar de leerlos todos.

flushBuffers: Vector de `cln_task_instance_argument` que contiene todos los argumentos que deben leerse.

numBuffers: Número de elementos de `flushBuffers`.

7.3. Gestión de recursos

Para poder ejecutar correctamente una tarea dentro de un dispositivo es necesario inicializar una serie de objetos de memoria de OpenCL. Cada objeto de memoria se corresponde a un argumento de tipo puntero en la función que queremos ejecutar. En el caso de los argumentos de tipos básicos (enteros, etc.) no se utilizan objetos de memoria. La *API* de OpenCL permite especificar si el objeto de memoria es de escritura (entrada), lectura (salida) o ambos (entrada/salida). Por otra parte, también es posible definir una política de memoria para el dispositivo. OpenCL define dos políticas:

1. Copiar los datos explícitamente a la memoria del dispositivo.

2. Utilizar directamente los datos de la memoria del procesador principal, con la posibilidad de que la implementación de OpenCL que usemos aplique técnicas de *cache* para acelerar la ejecución de tareas.

En nuestro caso hemos utilizado la política de copia explícita si el dispositivo es una *GPU*. En el caso de que estemos ejecutando la tarea sobre un dispositivo del tipo *CPU*, la política seguida es la segunda puesto que una *CPU* utiliza la memoria principal para almacenar los datos y por tanto, es posible ahorrar copias.

A parte de las políticas de gestión de memoria, se ha de decidir qué región de memoria se copiará exactamente al dispositivo. El tamaño de los datos a copiar así como su dirección base dependerá tanto de la tarea concreta que vayamos a ejecutar como del bucle que contiene la llamada.

A continuación utilizamos un ejemplo (Filtro FIR) para explicar cómo se deduce qué fragmentos de memoria se deben transferir al dispositivo.
Para deducirlo, se debe analizar tanto el espacio de iteraciones del NDRange como el rango de valores accedidos dentro de las tareas que son especificadas por el programador.

En el código 7.7 se muestra el código de la tarea así como el del programa principal que llama a la función a lo largo de un espacio de iteraciones (líneas 13 a 16).

```
#pragma omp target device(gpu)
#pragma omp task input(a[-KERN_LEN/2:KERN_LEN/2], kr[KERN_LEN])
    output(c[1])
void FIR(float *a, float *kr, float *c){
     *c = 0;
     int j,k;
     for (k=0, j = -KERN_LEN/2; j<=KERN_LEN/2; j++, k++){
          *c += a[j]*kr[k];
     }
}
...
int main(int argc, char *argv[])
{
     #pragma omp parallel for
     for (i=KERN_LEN/2; i<(SIG_LEN-(KERN_LEN/2)); i++){
          FIR(&a[i], &kr[0], &c[i]);
     }
...
}
```

Código 7.7: Implementación de un filtro FIR usando OmpSsCL.

En la línea 2 del código 7.7 se define que en cada ejecución de la función `FIR`, se usan los datos comprendidos entre -KERN_LEN/2 y KERN_LEN/2 respecto a la dirección del parámetro `a` que se pasa a la tarea. Esto lo define el programador en base a los accesos que se realizan en el bucle dentro de la tarea (línea 6 del código 7.7).

Asimismo, también se especifica que se usarán las KERN_LEN primeras posiciones a partir de la dirección `kr` que se pase como parámetro y que sólo se usará un elemento a partir de la dirección `c` pasada como parámetro.

Como podemos ver en el código 7.7, tanto `a` como `c` son accedidos por la variable de inducción, de modo que los datos copiados también dependerán del bucle que contenga la llamada a la función de la línea 15 del código 7.7.

De este modo, deberemos copiar SIG_LEN-KERN_LEN elementos de `a`, mas los KERN_LEN/2 y KERN_LEN/2 *extra* que se definen en el pragma del código 7.7. Es decir se deberá copiar el rango [*inicio del bucle* + (-KERN_LEN/2) .. *final del bucle* + KERN_LEN/2],

En el caso del parámetro `kr` hay que copiar los KERN_LEN primeros elementos independientemente el bucle que contiene la función, ya que este argumento no está accedido por la variable de inducción.

Para el parámetro `c`, debemos copiar SIG_LEN-KERN_LEN por las cotas del bucle ya que el programador ha definido en el pragma de la tarea que sólo se usa un elemento de este vector, o lo que es lo mismo [*inicio del bucle*+0 .. *final del bucle*+0], ya que `c[1]` equivale a `c[0:0]`.

El código 7.8 muestra el código OpenCL generado para la tarea `FIR` definida en el código 7.7. El acceso a los vectores dentro del kernel se realiza en función de la cota inferior del bucle (resultado de la llamada `get_global_id(0)`, en la línea 2 del código) y el desplazamiento relativo (`__cl_offset_arg...`), para cada vector, respecto a la dirección base que se pasa como parámetro del vector.

En desplazamiento relativo se calcula en función de la cota inferior del bucle, común a todos los vectores accedidos por la variable de inducción y la posición real que se debe acceder a cada vector, por ejemplo, en la primera ejecución de este kernel, `get_global_id(0)` retorna `KERN_LEN/2`, por tanto, para que el acceso a `c` en la línea 7 del código 7.8 sea

correcto, (necesitamos acceder a `c[0]` en la primera iteración) necesitamos un desplazamiento, en este caso `__cl_offset_arg_c`, que en este caso vale `-KERN_LEN/2`.

```
1  __kernel void convParallel(__global float *a, __global float *kr
       , __global float *c, int __cl_offset_arg_a, int
       __cl_offset_arg_kr, int __cl_offset_arg_c){
2  int __cl_local_i = get_global_id(0);
3  c[__cl_offset_arg_c + __cl_local_i] = 0;
4  int j, k;
5  for (k = 0 , j = -KERN_LEN / 2; j <= KERN_LEN / 2; j++ , k++){
6
7      c[__cl_offset_arg_c + __cl_local_i] += a[(j) +
           __cl_offset_arg_a + __cl_local_i] * kr[(k) +
           __cl_offset_arg_kr];
8
9  }
10 }
```

Código 7.8: Código *kernel* OpenCL generado a partir del código 7.7.

La alternativa a utilizar estos desplazamientos sería copiar el vector `c` entero, lo cual generaría mayor trafico de datos, que supone un impacto en el rendimiento mucho mayor al uso de los desplazamientos.

En general, la figura 7.1 nos ayuda a mostrar gráficamente el rango de datos que se copian según la definición de los argumentos en el *pragma* de la tarea especificada, suponiendo que la función de la tarea se llama para un espacio de iteraciones de 0 a N tal y como muestra el código 7.9. Los datos que se copiarán aparecen resaltados en azul en la figura. En los distintos apartados se muestra, a la izquierda, los datos de entrada

o salida que se especifican en la tarea. Debajo de cada una de las figuras, se muestra las cotas inferior y superior del bucle *for*, marcadas con *0* y *N* respectivamente. También se marca el *offset* o desplazamiento del vector con respecto a la cota inferior del bucle.

```
for(i=0; i<N; i++) {
    f(&v[i]);
}
```

Código 7.9: Bucle de ejemplo para determinar los rangos de datos que se deben copiar al dispositivo.

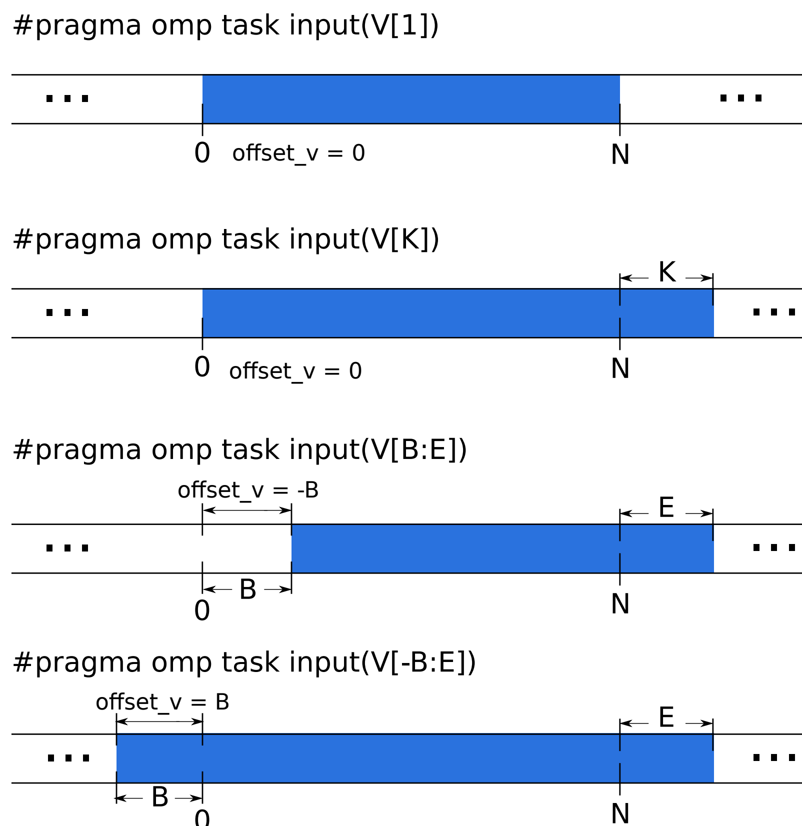

Figura 7.1: Ejemplos de rangos de datos a copiar a un dispositivo.

7.4. Optimización de transferencias de memoria

Durante las pruebas realizadas se detectó que uno de los mayores cuellos de botella en la ejecución de muchos programas eran las transferencias de memoria. Una posibilidad para ahorrar transferencias de memoria es mantener una serie de datos siempre en el dispositivo, de modo que se puedan aprovechar por más de una tarea (o varias ejecuciones de la misma tarea). La forma de especificar que un dato debe permanecer en el dispositivo es la cláusula `present`. Todos los argumentos que estén en esta directiva se mantendrán en el dispositivo, de manera que no sea necesario copiarlos repetidamente en caso de volverse a utilizar.

Por otro lado, los datos de salida que se encuentren en la directiva `present` no se leerán si no que permanecerán en el dispositivo. Esto implica que si queremos recuperar alguno de ellos al finalizar la ejecución de la tarea, tendríamos que ejecutar un `taskwait`, en el cual leeríamos todos los datos de todos los dispositivos. Para evitar tener que leer todos los datos al final de la ejecución de la tarea, añadimos la cláusula `flush`. La construcción `flush` ya existía en OpenMP y OmpSs y su función es garantizar que todos los datos afectados por esa construcción sean visibles a todos los threads. Aquí le hemos añadido la función que los datos afectados por esa construcción, y que se encuentran en un dispositivo de forma `present`, se copiarán a la memoria principal del *host* después de la ejecución de la tarea. De esta manera no es necesario leer todos los datos de los dispositivos si tan solo necesitamos recuperar algunos.

Capítulo 8

Instalación y uso

En esta sección se muestran los requisitos *software*, los pasos de instalación del sistema y cómo utilizarlo.

8.1. Requisitos del sistema

A continuación se muestran los requisitos que debe tener el sistema para poder instalar y utilizar correctamente nuestro entorno de compilación. Nótese que con el código fuente del sistema se distribuye un *script* (localizado en el directorio "`tools`") que automatiza todas las operaciones necesarias para instalar estos requisitos de OmpSsCL: Nanos, Mercurium, Extrae y Paraver.

Los requisitos previos a la instalación del entorno de ejecuciones son los siguientes:

- Sistema Operativo GNU/Linux.
- Bison 2.4.1 o posterior.
- flex 2.5 o posterior.
- gperf 3.0 o posterior.
- automake-1.9 o posterior.

- autoconf-2.59 o posterior.
- libtool 1.5.22 o posterior.
- gcc y g++ 4.1 o posterior.
- git (si se desea instalar Nanos y Mercurium desde los repositorios.
- Controladores de los dispositivos que soporten OpenCL.
- SDK de OpenCL.
- libxml2-dev (si se desea instrumentar el código de Nanos).
- graphviz y doxigen (si se quiere generar la documentación de Mercurium y Nanos).
- Mercurium 1.3.5.8 (o posterior).
- Nanos 0.6a.
- Extrae 2.1.1 (si se desea instrumentación).
- Paraver (para visualizar las trazas generadas por Extrae).

8.2. Instalación

8.2.1. Instalación de los requisitos

A continuación se describen los pasos necesarios para instalar Mercurium, Nanos, Extrae y Paraver.

Lo primero es obtener los programas:

```
1 mkdir $SOURCE_DOWNLOADS && cd $SOURCE_DOWNLOADS
2 git clone "http://pm.bsc.es/git/mcxx.git"
3 git clone "http://pm.bsc.es/git/nanox.git"
```

```
4 #para obtener los programas para la instrumentacion debemos
    ejecutar tambien los siguientes comandos
5 wget "http://www.bsc.es/ssl/apps/performanceTools/files/extrae
    -2.2.0.tar.gz"
6 wget "http://www.bsc.es/ssl/apps/performanceTools/files/
    wxparaver64.tar.gz"
```

La variable de entorno `$SOURCE-DOWNLOADS` representa el directorio donde se descargarán las herramientas para su posterior instalación.

Nótese que es posible que las últimas versiones de los repositorios no coincidan con los comandos mostrados.

Es posible también que las nuevas versiones de Nanos o Mercurium presenten alguna incompatibilidad con el *software* desarrollado en este proyecto. Para obtener las versiones exactas de Mercurium y Nanos que se han utilizado durante el desarrollo del proyecto es necesario ejecutar los siguientes comandos.

```
1 #Obtener versi n anterior de Mercurium
2 cd $SOURCE-DOWNLOADS/mcxx
3 git checkout 7c8ce37be8653b3b1122cfcffb089c080ad408e6
4 cd $SOURCE-DOWNLOADS/nanox
5 #Obtener version anterior de Nanos
6 git checkout 051503476b8f41c95b3a7f4618104cd749b4a42b
```

El siguiente paso es configurar, compilar e instalar las herramientas que acabamos de obtener. Si queremos activar la instrumentación antes de compilar Mercurium y Nanos, debemos instalar Extrae y opcionalmente Paraver para visualizar los datos:

Instalación de Extrae:

```
mkdir build
tar -xvf '$SOURCE_DOWNLOADS/extrae-2.2.0.tar.gz' -C build/
mkdir build/extrae && cd build/extrae
# $PREFIX_EXTRAE indica el lugar donde se instalara extrae
./configure \
    --prefix="$PREFIX_EXTRAE \  #Ruta de instalaci n de Extrae
    --enable-nanos \
    --enable-pthread \
    --without-papi \
    --without-mpi \
    --without-unwind \
    --without-dyninst \
    --enable-posix-clock
make && make install
```

Instalación de Paraver:

```
tar -xzvf "$SOURCE_DOWNLOADS/wxparaver64.tar.gz" -C /tmp
cp -r /tmp/wxparaver64/* -t $PREFIX_PARAVER
rm -f -r "/tmp/wxparaver64"
```

Antes de compilar e instalar Nanos, debemos realizar unas pequeñas modificaciones en el código. Debido a un conflicto con algunas versiones de la implementación de OpenCL de AMD, debemos desactivar la redefinición de los operadores *new* y *delete* en el código de gestión de memoria de Nanos. Esta operación se puede realizar con los siguientes comandos.

```
cd $SOURCE_DOWNLOADS/nanox
cp src/support/new.cpp src/support/new.cpp.backup
sed -i "s|^void|//void|g" src/support/new.cpp
```

Hecho esto podemos pasar a compilar Nanos:

```
cd $SOURCE_DOWNLOADS/nanox/
autoreconf --force --install
cd ../..
mkdir build/nanox
cd build/nanos
$SOURCE_DOWNLOADS/nanox/configure
    --prefix=$PREFIX_NANOS    #ruta de instalacion de Nanos
    --disable-gpu-arch \
    #$PREFIX_EXTRAE es la ruta donde hemos instalado extrae
    --with-extrae="$PREFIX_EXTRAE"
make
make install
```

Si quisiéramos generar la documentación de Nanos, tendríamos que ejecutar el comando `make doc`. En la configuración de Nanos, hemos deshabilitado el soporte para *GPU* que tiene Nanos, ya que interferiría en el funcionamiento de nuestro *runtime*.

Pasamos ahora a instalar Mercurium, de manera similar a como hemos procedido con Nanos, usando los siguientes comandos.

```
cd $SOURCE_DOWNLOADS/mcxx
```

```
2  autoreconf --force --install
3  cd ../..
4  mkdir build/mcxx && cd build/mcxx
5  $SOURCE_DOWNLOADS/mcxx/configure
6      --prefix="$PREFIX_MCXX" \ #ruta de instalacion de Mercurium
7      --enable-tl-openmp-nanox \
8      --enable-ompss \
9      #Ruta de la instalacion de Nanos
10     --with-nanox="$PREFIX_NANOS" \
11     --enable-tl-superscalar \
12     --with-superscalar-runtime-api-version=5 \
13 make
14 make install
```

De la misma manera que ocurría con Nanos, si queremos generar la documentación tenemos que ejecutar el comando `make doc`

Instalación de la fase de compilación que soporta OmpSsCL

Una vez instalados todos los paquetes necesarios, podemos proceder a la instalación de nuestro sistema. Para ello, el primer paso sera indicar dónde hemos instalado Mercurium y Nanos. Esto se hace modificando los ficheros *Makefile*, en el directorio "`mcxxCLN`". El *Makefile*, por defecto, tiene definido que Mercurium Y Nanos están instalados en el directorio "`./../tools/`" respecto al directorio donde se encuentra dicho fichero *Makefile*.

```
2  MERCURIUM_SOURCE_DIRECTORY=./../tools/source-downloads/mcxx
3  MERCURIUM_BUILD_DIRECTORY=./../tools/build/mcxx
```

```
4 MERCURIUM_INSTALL_DIRECTORY=./../tools/install2
```

Si se ha instalado Mercurium en otro directorio distinto, será necesario configurar estas variables convenientemente para que apunten a la ruta correcta.

Con los siguientes comandos incorporaremos la fase OmpSs to OpenCL a las fases de transformación de código de Mercurium.

Compilamos la fase del Mercurium usando los siguientes comandos:

```
1 cd mcxxCLN
2 make
3 make install
```

Generación de la librería del runtime de OmpSsCL

Para compilar el *runtime* correctamente será necesario especificar la ruta donde hemos instalado el *runtime* de OpenCL y todas las herramientas necesarias para su funcionamiento (Nanos, etc.). El código 8.1 muestra un ejemplo de la configuración de los directorios para el fichero Makefile del directorio "`OpenCLNanos/`", si se han instalado las herramientas en otra ruta, las variables deberán configurarse convenientemente.

```
3 OPEN_CL_INC=/opt/amd-app-sdk-v2.4-lnx64/include
4 OPEN_CL_LIB=/opt/amd-app-sdk-v2.4-lnx64/lib/x86_64
5
6 TOOLS_INSTALL_PATH=../tools/install2
7 NANOS_INC=$(TOOLS_INSTALL_PATH)/include/nanox #nanos include
    path
```

Código 8.1: Ejemplo de configuración del Makefile de OpenCLNanos

Procedemos de manera similar para compilar el *runtime*, en este caso no es necesario instalar nada, tan sólo generar la librería dinámica.

```
cd OpenCLNanos
make
```

8.3. Uso del sistema

El compilador se invoca usando el siguiente comando:

```
ssclncc <opciones> fichero [fichero...] -o salida
```

8.3.1. Opciones del compilador

Las opciones de compilación específicas para el entorno OmpSsCL son las siguientes:

- **- - disableNanos**: Deshabilita la fase de Nanos de Mercurium y por consiguiente su uso. De esta manera el código se ejecutará de manera secuencial puesto que usamos Nanos para la creación de las tareas, planificación y gestión de las dependencias. También permite obtener el código generado por nuestra fase de compilación.
- **- - variable=generate_report_on_taskwait:1**: Permite generar un pequeño informe que contiene estadísticas sobre la ejecución de las tareas cuando se realiza un *taskwait*. Utilizando esta opción podemos obtener datos sobre el tiempo invertido

en cada una de las operaciones ejecutadas en el runtime (transferencias de datos con los dispositivos, tiempo ejecución de los *kernels*, etc.)

Además de todas estas opciones, se pueden utilizar todas las opciones de Mercurium y de las fases que estén activadas en ese momento. Se puede obtener más información sobre las opciones utilizando el comando:

```
1 ssclncc --help
```

Ejemplo de uso

Con el siguiente comando obtenemos un fichero ejecutable para el *host* y una serie de ficheros de código OpenCL con extensión *.cl* que contienen las funciones que se ejecutarán en los dispositivos.

```
1 ssclncc program.c -o program
```

El nombre de un fichero OpenCL tiene la estructura <nombre del código de entrada>_<nombre de la función>-parallel_ssclncc.cl.

Una vez compilado el programa, podemos ejecutar el binario. Para ejecutarlo es necesario indicar el número de hilos de ejecución que utilizará el programa. Para esto definimos la variable de entorno `OMP_NUM_THREADS`. Esto se puede hacer justo antes de ejecutar el programa tal y como se muestra en el ejemplo a continuación.

```
1 OMP_NUM_THREADS=2 ./program [argumentos]
```

El número de *threads* aquí especificado afectará al número de tareas que se puedan ejecutar en paralelo. Cabe destacar que esto se verá limitado por el sistema en que se ejecute el programa. Por ejemplo, si tenemos una *GPU* instalada, sólo de podrá ejecutar una tarea para *GPU* en un momento dado.

Para ejecutar el programa con instrumentación, debemos compilar el código con la opción `instrumentation` como se muestra a continuación:

```
1 ssclncc program.c -o program --instrumentation
```

Para ejecutarlo, será necesario definir algunas variables de entorno:

```
1 NX_ARGS='--keep-mpits --instrumentation=extrae' OMP_NUM_THREADS
    =2 ./program [argumentos]
```

La variable `NX_ARGS` contiene los argumentos que se le pasan al runtime de Nanos. En este caso indicamos que los ficheros intermedios de instrumentación no se deben eliminar (`--keep-mpits`) y que queremos generar los datos de instrumentación en el formato de *extrae* (`--instrumentation=extrae`). Para facilitar esta tarea se ha creado el *run_with_extrae.sh* (disponible en el directorio "`tools`" de la distribución de código) que inicializa la variable de entorno `NX_ARGS` antes de ejecutar el programa. El script se invoca de la siguiente manera:

```
1 run_with_extrae.sh programa [argumentos del programa]
```

Capítulo 9

Resultados

En este capítulo se presentan los resultados obtenidos en las pruebas. En primer lugar se muestra el entorno de pruebas usado. A continuación se muestran los resultados obtenidos para las distintas aplicaciones analizadas.

De las tres aplicaciones que se analizan, dos de ellas (k-means y BFS) pertenecen al *benchmark* Rodinia, el filtro FIR se implementó durante el desarrollo del sistema y se incluye en los resultados por ser una aplicación ámpliamente utilizada en diversos campos.

Finalmente se muestra el comportamiento del sistema cuando se ejecutan tareas de manera concurrente con varios hilos de ejecución.

9.1. Entorno de pruebas

A continuación se describe el entorno de pruebas que se ha utilizado para la ejecución de las pruebas de rendimiento de los diferentes programa. En la tabla 9.1 se muestran las características principales del equipo de pruebas y en la tabla 9.2 se muestra información detallada sobre la *GPU* utilizada.

Para más información sobre el sistema, consultar las secciones A.1 (Información de entorno de OpenCL) y A.2 (Información del procesador) del apéndice.

Sistema Operativo	Ubuntu 11.04 - 2.6.38-14-generic x86_64 GNU/Linux
Compilador	gcc 4.5.2
Controlador GPU	NVIDIA 295.33
CPU	Intel Core2 E6600 a 2.40 Ghz (2 cores)
GPU	NVIDIA GeForce GTX 550 Ti
Memoria RAM	4GB DDR2-667
Disco duro	2×320GB a 7200 RPM

Tabla 9.1: Características del sistema de pruebas.

Modelo	NVIDIA GeForce GTX 550 Ti
CUDA cores	192
Memoria	1024 MB
Interfaz de memoria	192 bit
Tipo de bus	PCI Express x16
Velocidad del bus	2.5 GT/s

Tabla 9.2: Características de la GPU.

Todos los tiempos mostrados en este capítulo corresponden al tiempo *elapsed* perteneciente únicamente a la ejecución de los cálculos de las diferentes aplicaciones. Se excluyen los tiempos de inicialización, tales como lecturas de ficheros de datos o las escrituras de resultados.

Por otro lado los tiempos pertenecen a la media de 10 ejecuciones de la aplicación.

9.2. Filtro FIR

En esta aplicación se efectúa una operación de filtrado sobre una señal. Tanto la señal como la respuesta impulsional del filtro son generados aleatoriamente.

La figura 9.1 muestra el tiempo de ejecución de tres implementaciones distintas. La primera, etiquetada como `OmpSsCL CPU` en la figura 9.1, corresponde a la implementación de la aplicación con OmpSsCL utilizando como dispositivo de ejecución la *CPU*. La segunda, `OmpSsCL`, corresponde a una implementación usando OmpSsCL, en este caso ejecutando las tareas en una *GPU*. Finalmente, la etiqueta `OpenCL` hace referencia a la aplicación implementada utilizando directamente la *API* de OpenCL.

Los datos de la figura 9.1, que corresponden a los tiempos de la tabla 9.3, muestran que los tiempos utilizando una *GPU* (OmpSsCL y OpenCL) son notablemente menores que en una *CPU*. Pese a que en una *CPU* no es necesario realizar copias de memoria, vemos como es mucho más lenta para la ejecución de tareas en paralelo que una *GPU*. Esto es debido básicamente a la mayor capacidad de cálculo en paralelo de las *GPUs*. Por otro lado, vemos como los tiempos obtenidos con la implementación con OmpSsCL son muy similares a los obtenidos por la implementación realizada a mano utilizando OpenCL. Esto demuestra que nuestro sistema se aproxima a la implementación realizada manualmente, introduciendo poco *overhead* con el *runtime*.

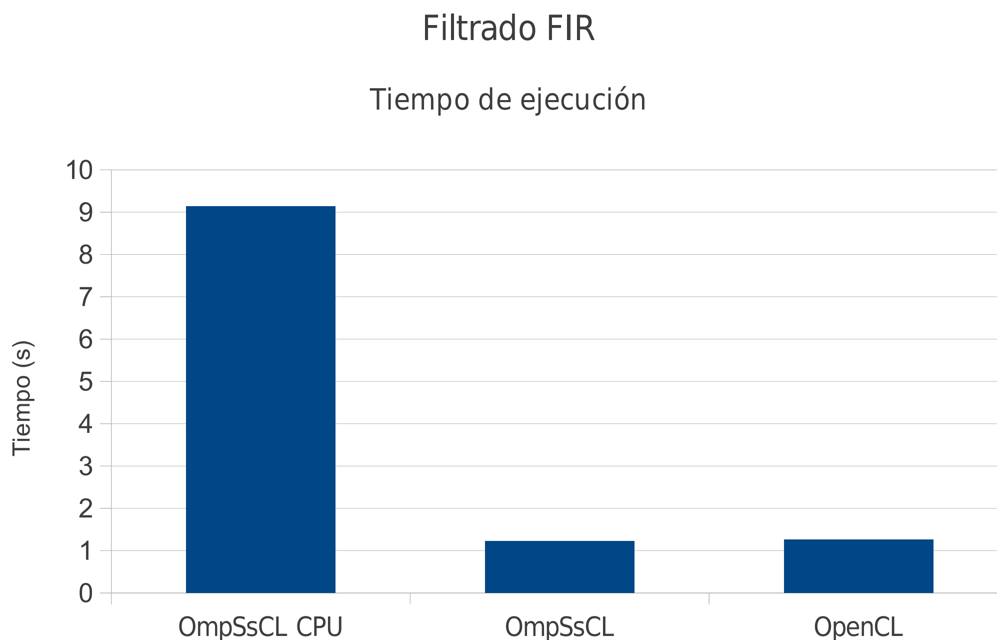

Figura 9.1: Tiempo de ejecución de las diferentes implementaciones del filtro FIR.

A continuación se analizará la implementación que usa nuestro entorno de ejecución OmpSsCL.

Implementación	Tiempo (s)
OmpSsCL CPU	9.137
OpenCL	1.255
OmpSsCL	1.221

Tabla 9.3: Tiempo de cálculo (*elapsed*) de las diferentes implementaciones del filtro FIR.

9.2.1. Análisis de la implementación OmpSsCL

El código 9.1 muestra la definición de la tarea y el código de llamada a la misma.

```
#pragma omp target device(gpu)
#pragma omp task input(a[-KERN_LEN/2:KERN_LEN/2], kr[KERN_LEN])
    output(c[1])
void FIR(float *a, float *kr, float *c){
    *c = 0;
    int j,k;
    for (k=0, j = -KERN_LEN/2; j<=KERN_LEN/2; j++, k++){
        *c += a[j]*kr[k];
    }
}
int main(int argc, char *argv[]){
...
    #pragma omp parallel for
    for (i=KERN_LEN/2; i<(SIG_LEN-(KERN_LEN/2)); i++){
        FIR(&a[i], &kr[0], &c[i]);
    }
...
}
```

Código 9.1: Implementación filtro FIR en OmpSsCL.

La figura 9.2 muestra la traza de una ejecución, visualizada con Paraver. En esta traza se puede ver el estado en el que se encuentra nuestro *runtime* durante la ejecución. Los colores de la traza determinan los estados del runtime en cada momento. La correspondencia color-estado se muestra en la figura 9.3. En la parte inferior de la traza se muestra el tiempo de inicio y final de la traza (en ns o ms). A lo largo de la sección se muestra únicamente la parte de la traza que corresponde a la ejecución del algoritmo que implementa cada aplicación.

Podemos observar en la figura 9.2 tres regiones. La primera, de color amarillo corresponde a la escritura de datos al dispositivo. La parte marcada de color rojo corresponde con la ejecución del *kernel* en el dispositivo y finalmente la región de color verde del final de la ejecución corresponde a la lectura de resultados.

Figura 9.2: *Zoom* de la traza de ejecución de la implementación de FIR con OmpSsCL.

- Fuera del runtime
- Inicialización de dispositivos
- Inicialización de tareas
- Gestión de tareas
- Espera del gestor de recursos
- Solicitus de recursos a OpenCL
- Liberación de recursos OpenCL
- Estcritura de datos al dispositivo
- Ejecución de kernel
- Lectura de datos del dispositivo

Figura 9.3: Leyenda de las trazas de ejecución.

La figura 9.4 muestra el porcentaje de tiempo empleado en cada estado del runtime. En la figura se observa como la ejecución del *kernel* ocupa la mayor parte del tiempo. Por

otro lado, la porción *otros* corresponde al tiempo empleado en efectuar gestiones como pedir o liberar recursos de OpenCL, esperar a que el dispositivo esté listo o inicializar escructuras de datos.

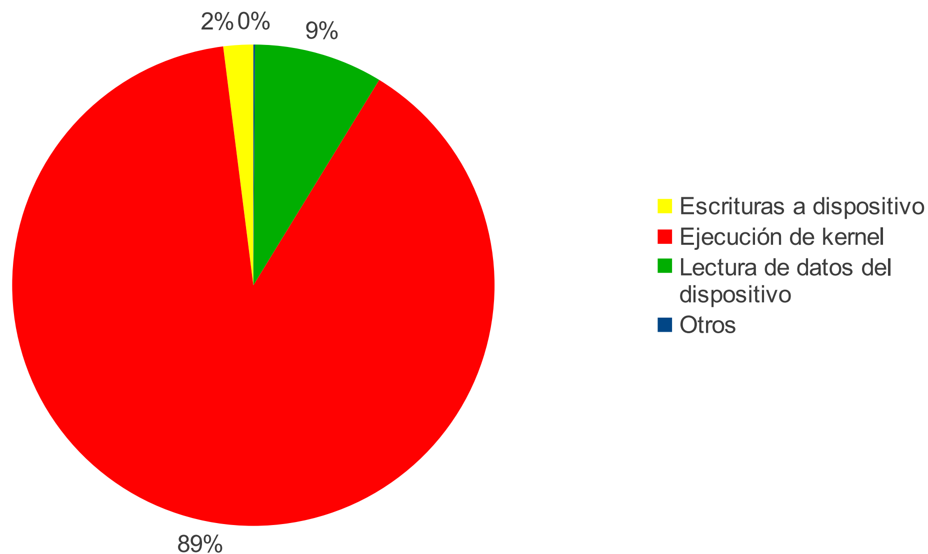

Figura 9.4: Porcentaje de tiempos de FIR empleado en el *runtime* de OmpSsCL.

9.3. K-means

K-means [7] es un algoritmo de clasificación (o *clustering*) utilizado en la minería de datos (entre otros campos) para separar un conjunto de observaciones en varios *clusters* según las características de estas observaciones.

La figura 9.5 muestra los tiempos de ejecución, detallados en la tabla 9.4, conseguidos con las diferentes implementaciones del mismo programa. La primera barra pertenece a la ejecución en la *CPU*, el resto pertenecen a ejecuciones efectuadas sobre una *GPU*. Estas versiones son:

OmpSsCL CPU: Implementación usando el *runtime* OmpSsCL, ejecutando las tareas

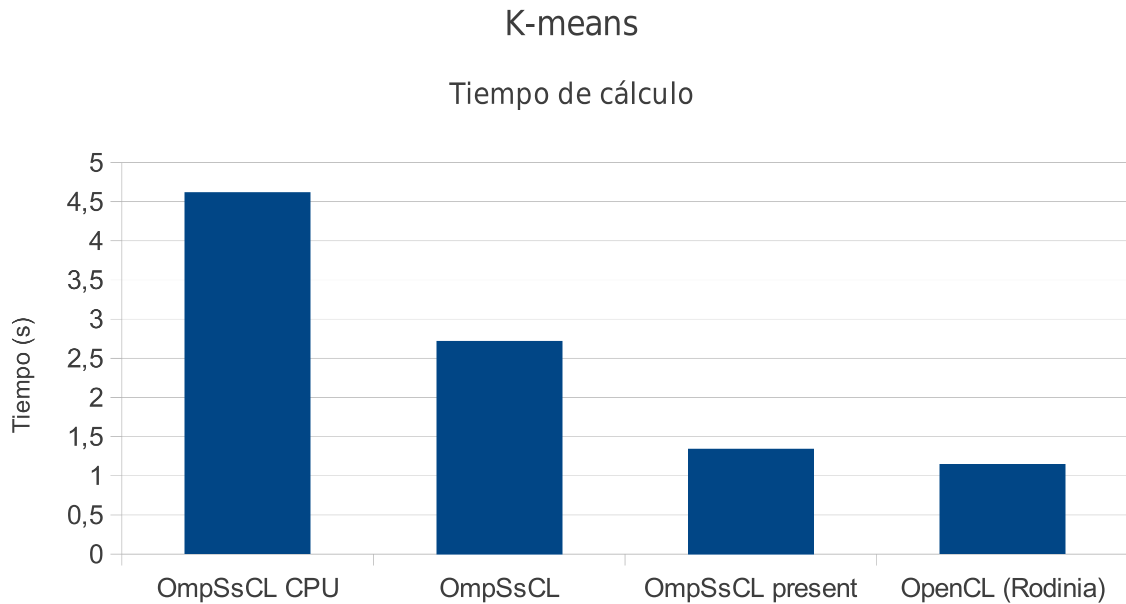

Figura 9.5: Tiempo de ejecución de las diferentes implementaciones de k-means.

Implementación	Tiempo (s)
OmpSsCL (CPU)	4.615
OmpSsCL (básico)	2.723
OmpSsCL (present)	1.346
OpenCL (Rodinia)	1.148

Tabla 9.4: Tiempos de ejecución de las diferentes implementaciones de k-means

en una *CPU*.

OmpSsCL: Implementación básica (sin optimizaciones) de las tareas usando OmpSsCL, ejecutándolas en una *GPU*.

OmpSsCL present: Similar a la anterior pero utilizando la cláusula present para ahorrar copias de datos al dispositivo.

OpenCL: Implementación utilizando directamente la *API* de OpenCL para acceder a los dispositivos, perteneciente al *Rodinia benchmark*.

Como se puede ver en la figura 9.5, la versión para *CPU* es notablemente más lenta que las versiones ejecutadas en la *GPU*. Esto es debido a que, pese a ahorrarnos las transferencias de datos con el dispositivo, la *CPU* no es capaz de explotar el paralelismo que presentan las tareas de manera eficiente en todos los núcleos disponibles en el sistema, tal y como lo hacen las *GPU*.

Por otro lado, se puede observar que la versión en que se usan las optimizaciones en las copias de datos (OmpSsCL present) es aproximadamente el doble de rápida que la versión básica (OmpSsCL), y presenta un rendimiento similar a la versión implementada directamente con OpenCL.

A continuación analizaremos en detalle las implementaciones que utilizan nuestro entorno OmpSsCL y se ejecutan en una *GPU*.

9.3.1. Análisis de la implementación básica OmpSsCL

El código 9.2 muestra el código de definición de la tarea de la implementación básica usando OmpSsCL. El código 9.3 muestra el bucle principal donde se ejecutan las tareas.

```
1  #pragma omp target device(gpu) present(feature)
2  #pragma omp task input(feature[feature_size], clusters[
       clusters_size], npoints, nclusters, nfeatures)\
3       output(membership[1])
4  void kmeans_task(float *feature, float *clusters, int *
       membership,
5                  int npoints, int nclusters, int nfeatures)
6  {
7       int i,index = 0;
8       float min_dist=FLT_MAX;
9       for (i=0; i < nclusters; i++) {
10          float dist = 0; float ans  = 0;
11          int l;
12          for (l=0; l<nfeatures; l++){
13              ans += (feature[l*npoints]-clusters[i*nfeatures+l])*
14                (feature[l * npoints]-clusters[i*nfeatures+l]);
15          }
16          dist = ans;
17          if (dist < min_dist) {
18              min_dist = dist;
19              index    = i;
20          }
21      }
22      *membership = index;
23 }
```

Código 9.2: Implementación básica de k-means en OmpSsCL.

```
do {
    int i, j, k;
    #pragma omp parallel for
    for (i=0; i<npoints; i++){
        kmeans_task(&feature_swap[i], clusters[0], &
            new_membership[i], npoints, nclusters, nfeatures);
    }
...
}while(delta<threashold);
```

Código 9.3: Código de llamada a la tareas `kmeans_task`.

La figura 9.6 muestra la traza de una ejecución del programa. En esta traza se han marcado las cuatro primeras iteraciones del bucle `do-while` del código 9.3. Nótese que cada iteración hará la ejecución de un bucle paralelo con una tarea como cuerpo de bucle, es decir, un *NDRange* en OpenCL.

Podemos observar que cada iteración consta de una copia a la memoria del dispositivo (amarillo), una ejecución del *kernel* (rojo) y una lectura de datos de la memoria del dispositivo (verde). De estos estados, el estado de copia a la memoria del dispositivo ocupa la mayor parte del tiempo de ejecución.

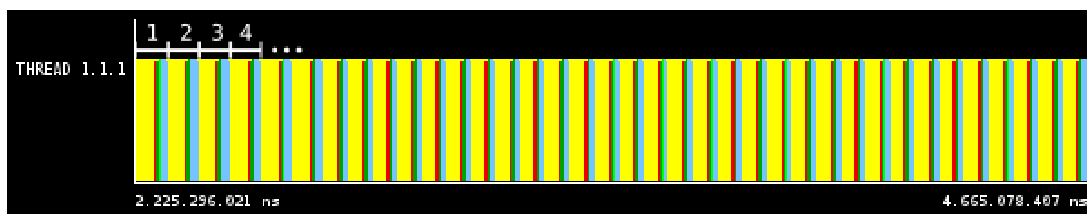

Figura 9.6: *Zoom* de una traza de ejcución de la omplementación básica de k-means con OmpSsCL.

En la figura 9.7 se muestra la fracción de tiempo que el runtime emplea en cada estado. Como se podía ya deducir en la traza (figura 9.6), la mayor parte del tiempo se emplea en escrituras de datos al dispositivo.

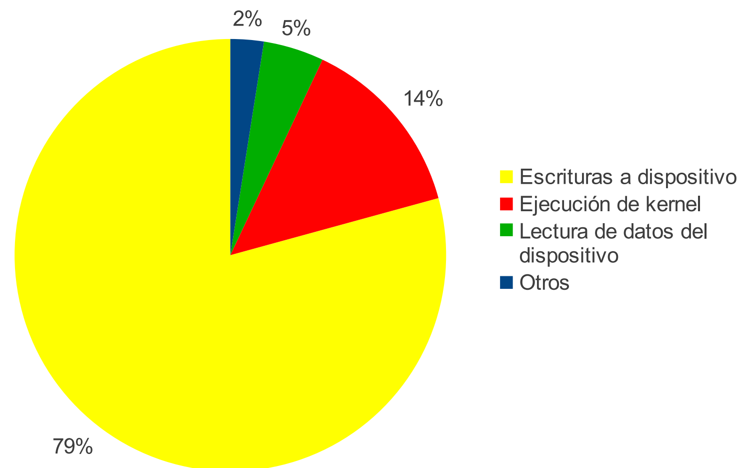

Figura 9.7: Porcentajes del tiempo de la implementación básica de k-means empleado en el runtime de OmpSsCL.

9.3.2. Análisis de la implementación OmpSsCL + present

Como se ha visto en el apartado anterior, una gran parte del tiempo se invierte en transferir datos al dispositivo. En esta versión se utiliza la cláusula `present` para indicar que los datos se mantengan en el dispositivo y no se copien si no es necesario. En el código 9.4 se muestra la nueva definición de la tarea. En la especificación del `target device` (línea 1 del código 9.4) se ha añadido que la variable `feature` debe considerarse `present` y por tanto, mantenerse en el dispositivo. El cuerpo de la tarea no ha variado y está definido en el código 9.2. El código de llamada a la tarea tampoco sufre ninguna modificación, se muestra en el código 9.3.

```
1 #pragma omp target device(gpu) present(feature)
```

```
2 #pragma omp task input(feature[feature_size], clusters[
    clusters_size], npoints, nclusters, nfeatures) output(
    membership[1])
3 void kmeans_task(float *feature, float *clusters, int *
    membership, int npoints, int nclusters, int nfeatures)
4 {
5 ...
6 }
```

Código 9.4: Implementación OmpSsCL de k-means usando la cláusula `present`.

En la traza de la ejecución del programa (figura 9.8) se puede apreciar que el tiempo dedicado a copias de datos (en color amarillo) ha disminuido significativamente. Se han marcado las cuatro primeras iteraciones del bucle `do-while` del código 9.3. La primera iteración ocupa mucho más tiempo que las demás puesto que se realiza la copia inicial de los datos. Esta copia corresponde a la región etiquetada como "A".
Las zonas de color azul pertenecen a operaciones que realiza el programa fuera de nuestro runtime, concretamente corresponden a una reducción efectuada por *CPU*.

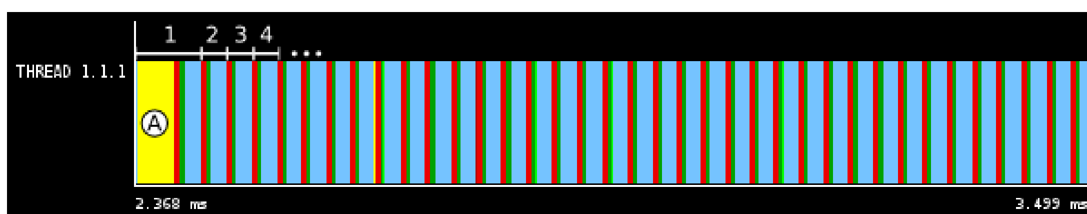

Figura 9.8: *Zoom* de una traza de ejecución de k-means en OmpSsCL con la cláusula `present`.

La figura 9.9 compara el tiempo absoluto y las proporciones de tiempo invertidas en cada uno de los estados. El tiempo dedicado a las copias de datos ha disminuido considerablemente respecto al caso anterior. Tal como puede verse en el gráfico etiquetado

con k-means present, ahora se dedica la mayor parte del tiempo consumido en nuestro runtime a ejecutar las tareas y no a realizar copias de datos. Esto tiene como consecuencia el hecho de que el tiempo de ejecución con la mejora *present* representa, aproximadamente, la mitad que en la implementación básica.

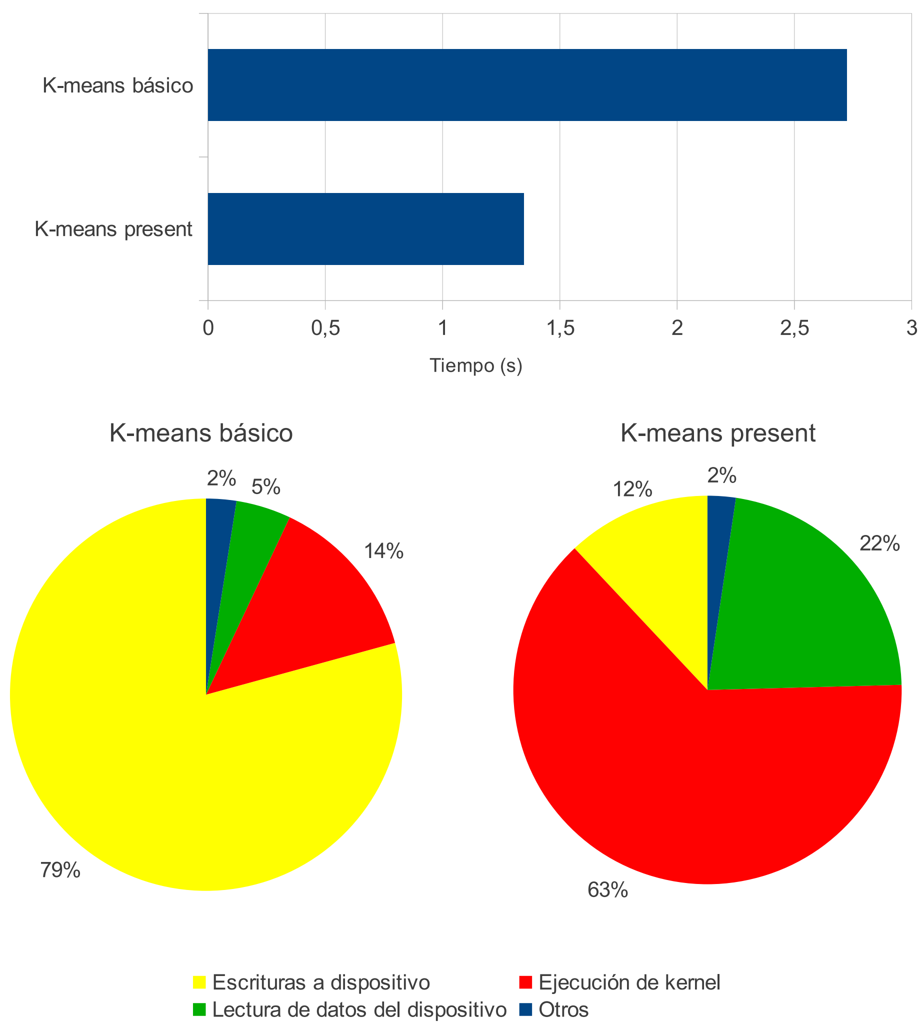

Figura 9.9: Comparativa de tiempos de ejecución (arriba) y del porcentaje en los diferentes estados de ejecución (abajo) k-means usando o no la cláusula `present`.

9.4. BFS

BFS (del inglés *Breadth-First Search*) efectúa una búsqueda en anchura a través de un grafo. De manera similar a k-means, se trata de un algoritmo iterativo en el que se efectúan varias iteraciones sobre el grafo de entrada. A diferencia de k-means, en este caso se ejecutan dos *kernels* en cada iteración.

La figura 9.10 muestra los tiempos de ejecución de BFS utilizando OmpSsCL sobre una *CPU* o una *GPU*, usando las diferentes funcionalidades de OmpSsCL para mejorar el rendimiento desde el punto de vista de las transferencias de datos. También se muestra la implementación de BFS utilizando directamente OpenCL (implementación extraída del Rodinia *benchmark*).

La primera barra corresponde a la ejecución de la aplicación utilizando la *CPU*, en todas las demás se ejecuta sobre una *GPU*. Los tiempos, mostrados también en la tabla 9.5, son *elapsed* time. Cada una de las barras corresponde al tiempo empleado en la ejecución de los *kernels*, las transferencias de memoria y otras tareas de gestión de recursos que efectúa el *runtime*.

Versión	**Tiempo (s)**
OmpSsCL CPU	0.196
OmpSsCL (sin optimizaciones)	0.517
OmpSsCL present (W)	0.293
OmpSsCL present (W+R)	0.131
OmpSsCL present + flush	0.061
OpenCL (Rodinia)	0.080

Tabla 9.5: Tiempos de ejecución de las diferentes implementaciones de BFS.

OmpSsCL (CPU): Implementación con OmpSsCL usando la *CPU* para la ejecución de tareas.

OmpSsCL: Implementación usando OmpSsCL, sin utilizar la cláusula `present` para optimizar las transferencias de memoria.

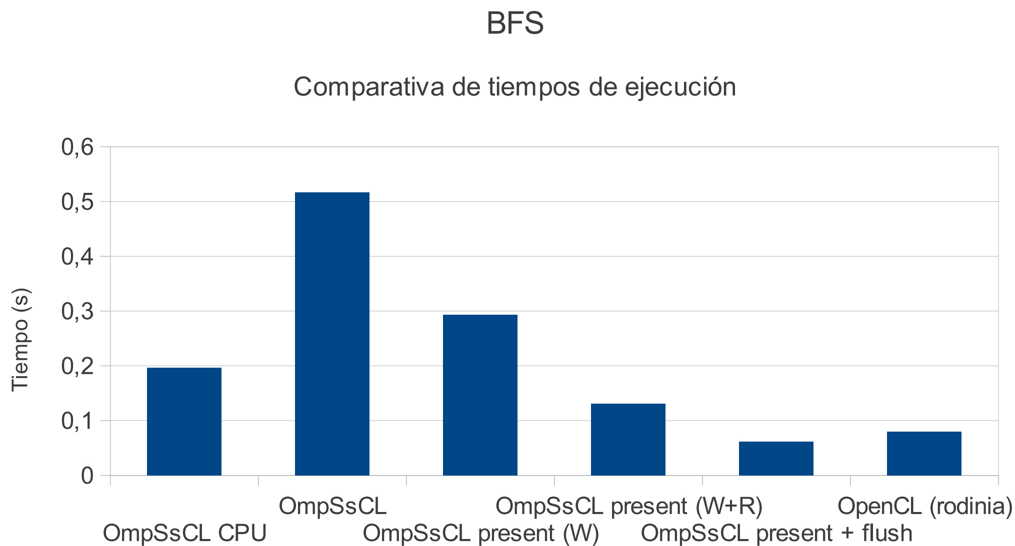

Figura 9.10: Comparativa de tiempos de las diferentes versiones de BFS.

OmpSsCL present (W): Implementación de BFS utilizando la cáusula `present` para optimizar las transferencias de memoria de escritura a los dispositivos, en este caso sólo se aplican las optimizaciones sobre las escrituras de datos a los dispositivos.

OmpSsCL present (W+R): Implementación de BFS utilizando la cláusula present para optimizar las transferencias de memoria de escritura y lectura a/de los dispositivos.

OmpSsCL present + flush: Implementación de BFS utilizando la cáusula `present` y la construcción flush para optimizar la transferencia a/de los dispositivos, y reducir los datos de los dispositivos, respectivamente.

OpenCL: Implementación de BFS usando OpenCL, perteneciente al Rodinia *benchmark*.

En la figura 9.10 podemos observar que el tiempo de ejecución para la versión OmpSs-CL en una *CPU* (primera barra) es menor que el de la versión básica del BFS con OmpSs-

CL sobre una *GPU* (segunda barra). Este es debido básicamente al número de excesivo de transferencias de datos que se realizan entre la memoria principal del *host* y la memoria de la *GPU*. El resto de versiones OmpSsCL, incluyen optimizaciones para reducir el numero de transferencias, que consiguen igualar y superar el rendimiento de la aplicación ejecutándose sobre *CPU*, tal y como se puede ver en la tabla 9.5.

A continuación se analizan las distintas implementaciones que se han realizado de BFS.

9.4.1. Análisis de la implementación básica usando OmpSsCL

En esta implementación no se ha utilizado la cláusula `present` ni la construcción `flush` para optimizar las copias de datos entre la memoria principal del *host* y la memoria de los dispositivos.

El código 9.5 muestra la definición de las tareas y en el código 9.6 se muestra el código donde se efectúan las llamadas a las tareas.

```
1 #pragma omp target device(gpu)
2 #pragma omp task input(h_graph_starting[1], h_graph_no_edges[1],
      h_graph_edges[edge_list_size], h_graph_visited[no_of_nodes],
      h_cost_c[1]) inout(h_cost[no_of_nodes],
     h_updating_graph_mask[no_of_nodes], h_graph_mask[1])
3 void bfs_1(int *h_graph_starting, int *h_graph_no_edges, int*
     h_graph_edges, int* h_graph_mask, int *h_updating_graph_mask,
      int* h_graph_visited, int *h_cost_c, int* h_cost){
4     if (*h_graph_mask == 1){
5         *h_graph_mask=0;
6         int i;
7         for(i=*h_graph_starting; i<(*h_graph_no_edges+*
```

```
            h_graph_starting); i++){
8               int id = h_graph_edges[i];
9               if(!h_graph_visited[id])
10              {
11                  h_cost[id]=*h_cost_c+1;
12                  h_updating_graph_mask[id]=1;
13  }   }   }   }
14
15  #pragma omp target device(gpu)
16  #pragma omp task inout(h_updating_graph_mask[1], h_graph_mask
        [1], h_graph_visited[1], stop[1])
17  void bfs_2(int* h_updating_graph_mask, int *h_graph_mask, int *
        h_graph_visited, int *stop){
18      if (*h_updating_graph_mask == 1){
19          *h_graph_mask=1; *h_graph_visited=1; *
                h_updating_graph_mask=0;
20          *stop=1;
21  }   }
```

Código 9.5: Definición de las tareas para la implementación con OmpSsCL para BFS.

```
1   ...
2   do{
3       ...
4       #pragma omp parallel for
5       for(i = 0; i < no_of_nodes; i++ )
6       {
```

```
7            bfs_1 (...);
8        }
9        #pragma omp parallel for
10       for(tid=0; tid< no_of_nodes ; tid++ )
11       {
12           bfs_2 (...);
13       }
14       #pragma omp taskwait
15       ...
16   }while(stop);
17   ...
```

Código 9.6: Código OmpSsCL de llamada a las tareas de BFS.

La figura 9.11 muestra una traza de la ejecución de esta versión de BFS. En esta traza se puede observar que la mayor parte del tiempo se invierte en las copias de datos al dispositivo (regiones amarillas en la figura 9.11).

En esta traza se han etiquetado las cuatro primeras iteraciones del algoritmo. Podemos ver como existe un patrón que se repite en cada iteración del algoritmo: se ejecutan primero un *kernel* (etiqueta "A") y después el segundo (etiqueta "B"). En este caso, cada ejecución de *kernel* lleva asociadas una serie de escrituras de datos al dispositivo (regiones amarillas) y algunas lecturas (regiones verdes), mientras el tiempo de ejecución del kernel corresponden a las regiones de color rojo.

La figura 9.12 nos muestra el tiempo invertido en la lectura, escritura, ejecución de *kernels* y otros estados. Como podemos observar, las transferencias de memoria suponen un 90 % del tiempo empleado en el *runtime*.

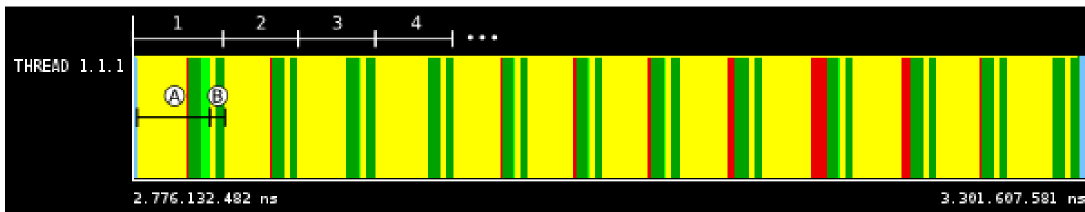

Figura 9.11: *Zoom* de la traza de ejecución de la versión sin optimizaciones de BFS.

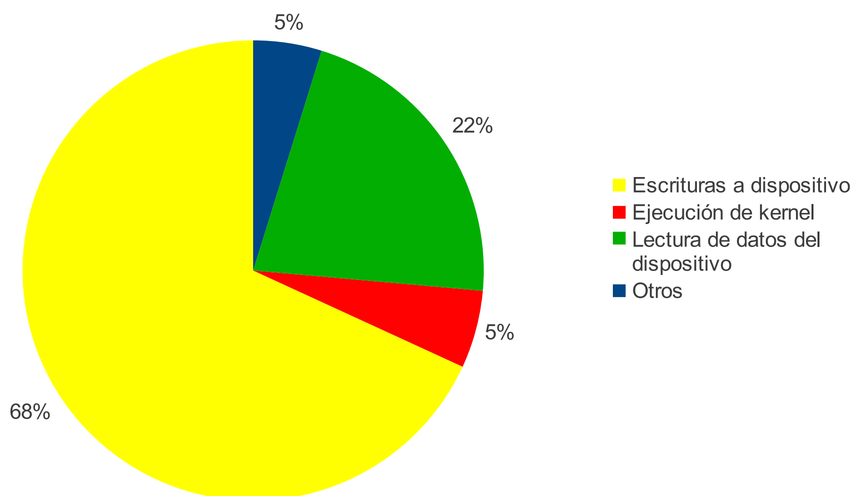

Figura 9.12: Fracción de tiempo de ejecución en los diferentes estados del runtime OmpSs-CL de la implementación BFS con OmpSsCL.

9.4.2. Análisis de la implementación OmpSsCL usando present (W)

En el código 9.7 se muestran las definiciones de las tareas para esta versión que incorporan la cláusula `present`. El código de las llamadas a las tareas no ha cambiado, y corresponde al código 9.6. El código del cuerpo de las funciones tampoco ha variado y puede consultarse en el código 9.5.

En esta versión, tal como se muestra en el código 9.7 se indica que las variables `h_graph_starting`, `h_graph_no_edges` y `h_graph_edges` se almacenen de forma permanente en el dispositivo mediante la cláusula `present` (línea 1 del código 9.7). De esta forma se evita tener que hacer las transferencias de copia de datos hacia el dispositivo en

cada iteración.

En esta implementación sólo se especifican como *present* las variables que son de entrada (figuran como `input` en la línea 2 del código). Concretamente con esto se evita la transferencia del grafo a cada iteración del bucle.

```
1  #pragma omp target device(gpu) present(h_graph_starting,
       h_graph_no_edges, h_graph_edges)
2  #pragma omp task input(h_graph_starting[1], h_graph_no_edges[1],
        h_graph_edges[edge_list_size], h_graph_visited[no_of_nodes])
        inout(h_cost[no_of_nodes], h_cost_c[1],
       h_updating_graph_mask[no_of_nodes], h_graph_mask[1])
3  void bfs_1(int *h_graph_starting, int *h_graph_no_edges , int*
       h_graph_edges, char* h_graph_mask, char *
       h_updating_graph_mask,
4          char* h_graph_visited, int *h_cost_c, int* h_cost)
5  {
6  ...
7  }
8
9  #pragma omp target device(gpu)
10 #pragma omp task inout(h_updating_graph_mask[1], h_graph_mask
       [1], h_graph_visited[1], stop[1])
11 void bfs_2(char* h_updating_graph_mask, char *h_graph_mask, char
        *h_graph_visited, int *stop){
12 ...
13 }
```

Código 9.7: Código OmpSsCL de las tareas de BFS utilizando la cláusula present para las variables de entrada (versión BFS present (w)).

En la figura 9.13 se muestra la traza de la ejecución de BFS para esta versión. En ella se han marcado las cuatro primeras iteraciones. Podemos ver como en la primera iteración (1) se realiza la copia del grafo (correspondiente a la región "A" de la figura 9.13) mientras que en las siguientes iteraciones estos datos no se vuelven a copiar. También podemos observar como la ejecución de los dos *kernels* siguen teniendo asociadas una serie de transferencias de datos, porciones "B" y "C" de la traza. Observamos también como los patrones de ejecución de los *kernels* se repiten a lo largo de todas las iteraciones.

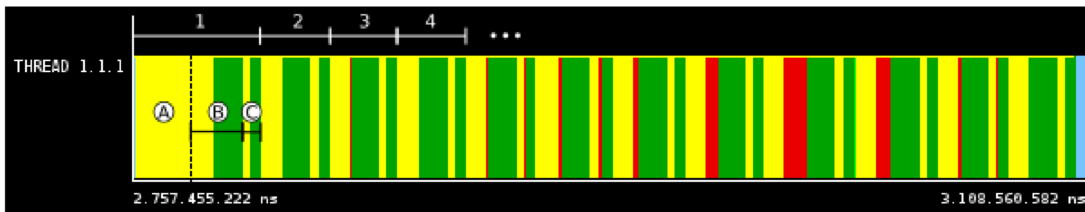

Figura 9.13: *Zoom* de la traza de ejecución de la implementación de BFS con optimizaciones en la escritura de datos (present (W)).

Con esta reducción de transferencias hemos logrado reducir el tiempo de BFS usando OmpSsCL sobre una *GPU* a casi la mitad. La figura 9.14 muestra una comparativa de los tiempos absolutos de los *kernels* de BFS usando o no la cláusula `present`.

También mostramos una comparativa del tanto por ciento que se invierte en cada estado de ejecución para las dos implementaciones: sin y con la cláusula `present`. Como se puede observar, hemos reducido significativamente el tiempo de escritura en el dispositivo.

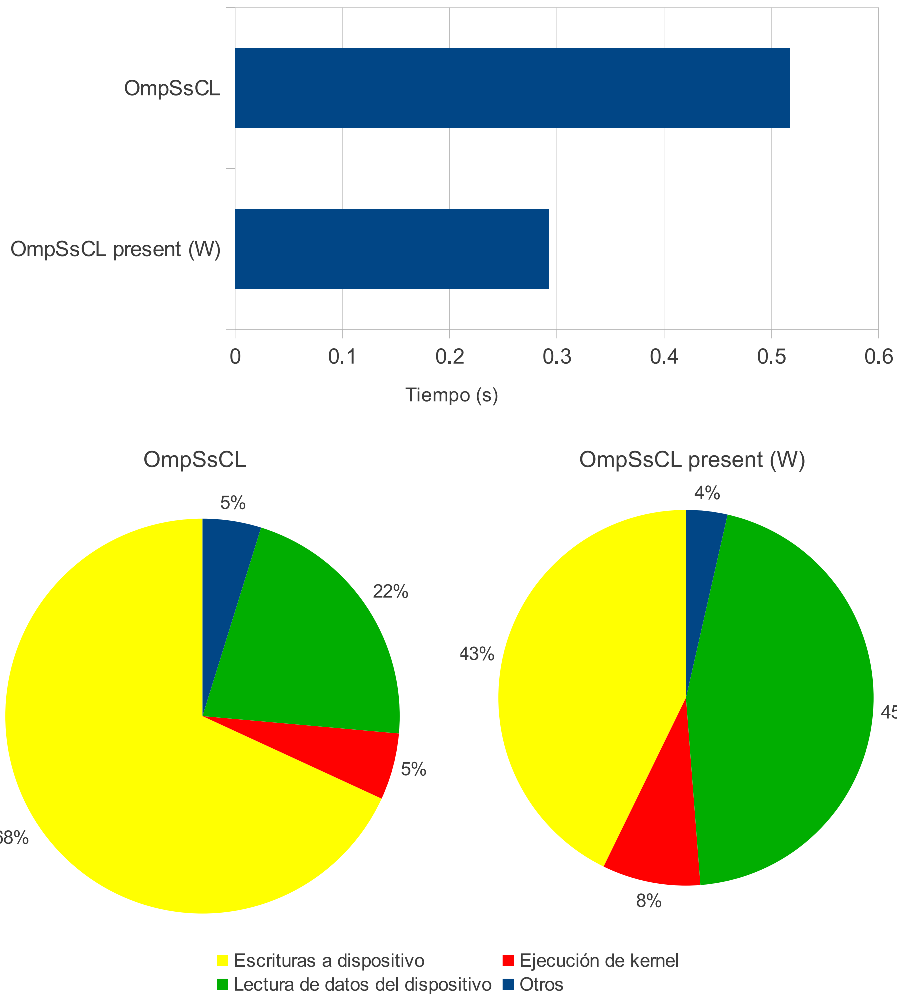

Figura 9.14: Comparativa de tiempos de ejecución (arriba) y del porcentaje en los diferentes estados de ejecución (abajo) de BFS usando o no la cláusula `present` para las variables de escritura (OmpSsCL y OmpSsCL present (W)).

9.4.3. Análisis de la implementación OmpSsCL usando present (W+R)

En este caso, además de la reducción de transferencias en las variables de entrada, se añaden las variables de entrada/salida al conjunto de argumentos `present`.

El código 9.8 muestra el código anterior con la especificación de `present` para las variables de entrada/salida (líneas 2 y 10 del código). Con esta nueva declaración, todas las variables menos la variable `stop`, se dejan en el dispositivo de manera permanente de

tal forma que la tarea `bfs_2` puede reutilizar los datos calculados por la tarea `bfs_1` sin necesidad de pasar por la memoria principal del *host*.

Sin embargo, la variable `stop`, generada por la tarea `bfs_2`, se debe conocer para saber si se debe realizar otra iteración del bucle `do-while(stop)` (código 9.6). Esto obliga a tener que forzar la escritura de `stop`, y del resto de las variables a la memoria principal del `host`. Esto se realiza con la directiva `#pragma omp taskwait` (línea 14 del código 9.6).

Como se puede ver en las líneas 2 y 10 del código 9.8, todas las variables menos `stop` se almacenan permanentemente en el dispositivo, de modo que no será necesario transferir estas variables entre el *host* y el dispositivo durante la ejecución del algoritmo.

No obstante, en el código de llamada a las tareas todavía conserva un `#pragma omp taskwait`, necesario para hacer la sincronización de datos y leer la variable `stop` (línea 14 del código 9.6).

```
1 #pragma omp target device(gpu) \
2     present(h_graph_starting, h_graph_no_edges, h_graph_edges,
        h_updating_graph_mask, h_graph_mask, h_graph_visited,
        h_cost, h_cost_c)
3 #pragma omp task input(h_graph_starting[1], h_graph_no_edges[1],
     h_graph_edges[edge_list_size], h_graph_visited[no_of_nodes])
     inout(h_cost[no_of_nodes], h_cost_c[1],
    h_updating_graph_mask[no_of_nodes], h_graph_mask[1])
4 void bfs_1(int *h_graph_starting, int *h_graph_no_edges , int*
    h_graph_edges, char* h_graph_mask, char *
    h_updating_graph_mask,
5         char* h_graph_visited, int *h_cost_c, int* h_cost
6 {
```

```
7 ...
8 }
9
10 #pragma omp target device(gpu) present(h_updating_graph_mask,
       h_graph_mask, h_graph_visited)
11 #pragma omp task inout(h_updating_graph_mask[1], h_graph_mask
       [1], h_graph_visited[1], stop[1])
12 void bfs_2(char* h_updating_graph_mask, char *h_graph_mask, char
        *h_graph_visited, int *stop)
13 {
14 ...
```

Código 9.8: Código OmpSsCL de las tareas de BFS utilizando la cláusula present para las variables de entrada (versión BFS present (W+R)).

La figura 9.15 muestra la traza de una ejecución de BFS con las nuevas cláusulas `present` que hemos añadido para las variables de lectura/escritura. En ella se han marcado las primeras iteraciones. En la primera de ellas (1) se puede observar que se realiza una gran copia de datos al dispositivo (región amarilla, (A)) correspondiente a los datos del grafo de entrada del problema.

En las sucesivas iteraciones, (2, 3, 4 y sucesivas), sin embargo, ya no se efectúan tantas escrituras (regiones de color amarillo) entre iteración e iteración como se podía observar en la traza de la figura 9.13. Esto es debido a que las variables de lectura/escritura no se escriben una y otra vez en el dispositivo ya que hemos especificado que están presentes en el mismo.

Por otro lado, vemos como al final de cada iteración se efectúa una lectura de datos (regiones de color verde). Éstas corresponden a la lectura de todos los datos que haya almacenados en el dispositivo. Esto es debido a que para sincronizar correctamente una

tarea, en esta versión, es necesario efectuar un `taskwait` que sincroniza todos los datos.

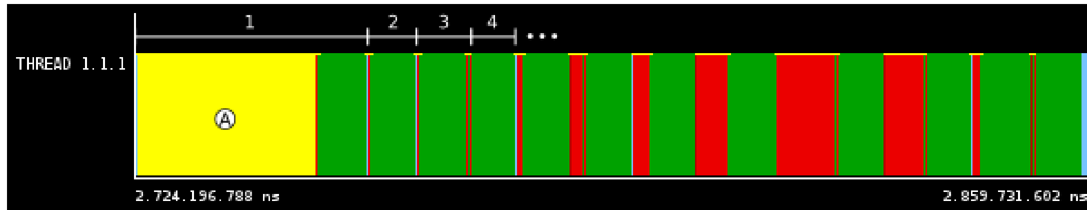

Figura 9.15: *Zoom* de la traza de ejecución de BFS con optimizaciones en la lectura y escritura de datos (present (W+R)).

En la figura 9.16 podemos ver como el tiempo de escrituras de datos ha descendido considerablemente, esto es debido principalmente a que los parámetros de entrada/salida de la tarea se mantienen en el dispositivo. El tiempo dedicado a la lectura de datos también se ha reducido, aunque la reducción ha sido menor que la reducción de escrituras y por tanto ocupa una mayor fracción del total. Como consecuencia de la reducción del tiempo empleado en transferencias de memoria (tanto lectura como escritura), el tiempo de ejecución del *kernel* también ocupa una mayor fracción del tiempo total. Observamos también en el gráfico de barras de la parte superior de la figura 9.16 como el tiempo de ejecución se ha reducido a menos de la mitad.

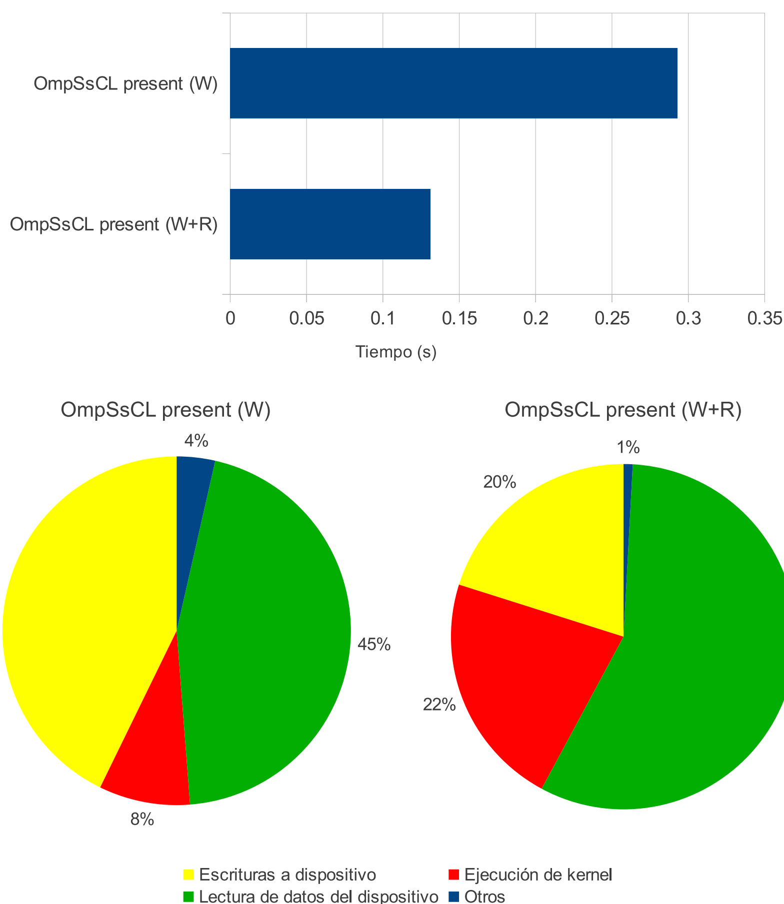

Figura 9.16: Comparativa de tiempos de ejecución (arriba) y del porcentaje en los diferentes estados de ejecución (abajo) de BFS usando o no la cláusula `present` para las variables lectura y escritura (OmpSsCL present (W) y OmpSsCL present (W+R)).

9.4.4. Análisis de la implementación OmpSsCL con present + flush

En esta versión añadimos el uso de la construcción `flush`. Esto provoca que después de ejecutar cada *kernel* se fuerce la lectura de los argumentos de la tarea que figuren en la construcción `flush`.

También se añade la cláusula `nowait`, que indica que no queremos que se realice sincronización alguna después de ejecutar la tarea. En este caso, Nanos sincronizará la ejecución en base a las dependencias de datos de las tareas generadas.

En esta implementación, el código de definición de las tareas de la versión anterior (código 9.8) no se modifica. El código 9.9 muestra cómo varía el código que llama a las tareas para especificar qué datos son los que se deben sincronizar. En este código hemos añadido la cláusula `nowait` a las construcciones `omp parallel for` (líneas 6 y 10). También se ha añadido la construcción `flush(stop)` en la línea 13 del código 9.9 para que `stop` se actualice en la memoria principaldel *host* y asi poder comprobar su valor en la condición del bucle `do-while`.

El código 9.9, muestra el código de llamada a las tareas (la definición de las tareas es idéntica a la realizada en el código 9.8. Nótese como aparecen las cláusulas `nowait` acompañando a las construcciones `omp for` (líneas 6 y 10). también se ha añadido la cláusula `omp flush` en la línea 13 para que la variable `stop` sea visible por el *thread* principal una vez ha finalizado la ejecución de la tarea.

```
1  #pragma omp parallel
2  {
3      do    {
4      #pragma omp single
5      stop=0;
6      #pragma omp for nowait
7      for(i = 0; i < no_of_nodes; i++ ){
8          bfs_1(...);
9      }
10     #pragma omp for nowait
11     for(tid=0; tid< no_of_nodes ; tid++ ){
12         bfs_2(...  , &stop);
13         #pragma omp flush(stop)
14     }
```

```
15    #pragma omp single
16    k++;
17    }while(stop);
18    #pragma omp taskwait
19 }
```

Código 9.9: Código de llamada a las tareas de la versión OmpSsCL present+flush de BFS.

La figura 9.17 muestra la traza de una ejecución de BFS usando la construcción `flush`. En la traza identificamos tres regiones: “A”, “B” y “C” separadas por líneas blancas.

Al principio de la ejecución se observa una gran región amarilla (A) que corresponde a la copia de datos al dispositivo. Después de la copia inicial, vemos que no se realizan grandes transferencias de datos (sólo las necesarias para leer la variable `stop`) durante la ejecución de las tareas (B).

Finalmente, se observa una gran región de color verde (C) que corresponde a la lectura de los resultados una vez acabada la ejecución de todas las iteraciones del algoritmo (B).

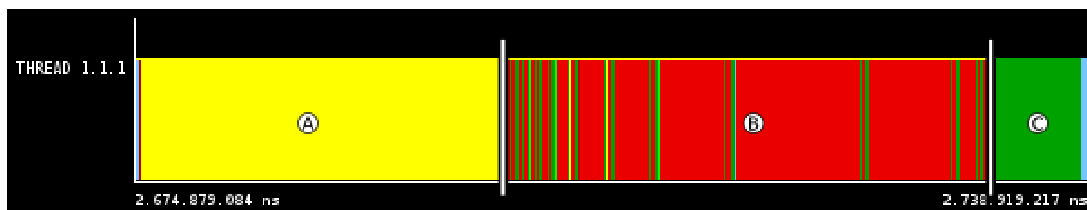

Figura 9.17: *Zoom* de la traza de ejecución de la implementación de BFS utilizando la cláusula `flush`.

En la figura 9.18 se muestra la comparativa de las versiones OmpSsCL present (W+R) y OmpSsCL present + flush. En ella se puede ver cómo el tiempo de ejecución del algoritmo se ha reducido a (aproximadamente) la mitad. También podemos observar en la figura como el tiempo dedicado a leer datos del dispositivo a disminuido considerablemente ya que sólo se leen los resultados al final de la ejecución del algoritmo.

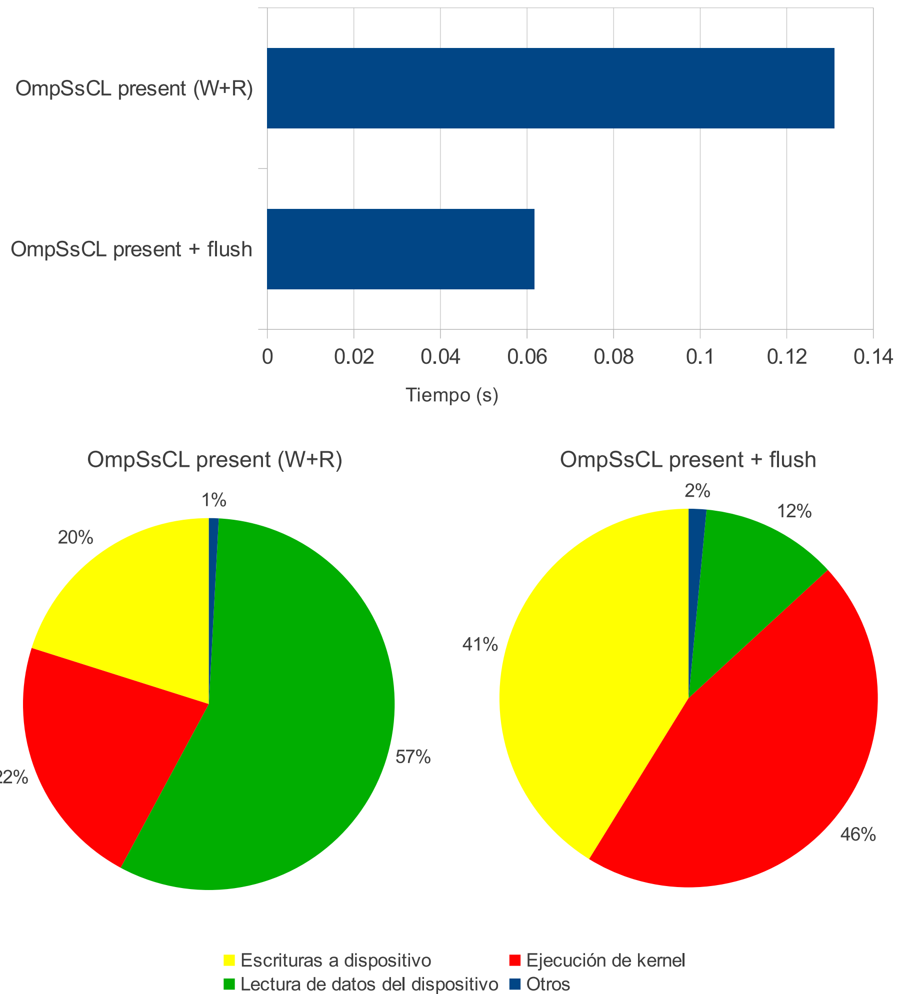

Figura 9.18: Comparativa de tiempos de ejecución (arriba) y del porcentaje en los diferentes estados de ejecución (abajo) de BFS usando o no la construcción `flush` (OmpSsCL present (W+R) y OmpSsCL present + flush).

9.5. Comparativa de rendimiento

A continuación se muestra una comparativa del rendimiento de las aplicaciones implementadas usando OpenCL y OmpSsCL.

La figura 9.19 muestra el *speedup* de cada implementación respecto a la ejecución secuencial del programa original en una *CPU* [1]. En el gráfico se muestran datos para

[1]Intel Core2Duo E6600 (1 core)

tres conjuntos de datos de entrada: A, B y C que contienen distintas instancias de los problemas que resuelve cada aplicación.

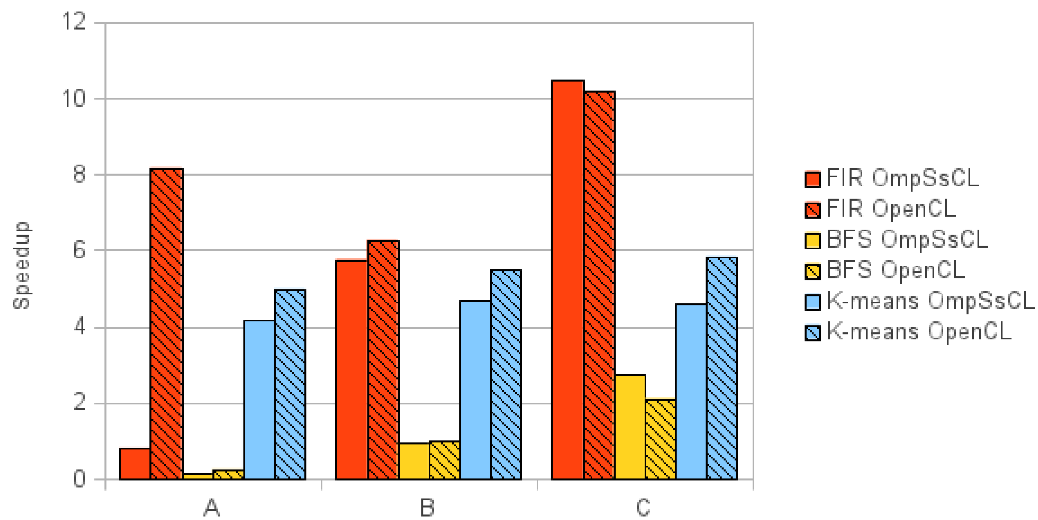

Figura 9.19: Comparativa del *speedup* de las aplicaciones programadas con OmpSsCL y OpenCL respecto a la ejecución secuencial en el procesador del *host*.

La tabla 9.6 muestra los tamaños de los datos de entrada para cada aplicación. Como se puede observar en la tabla, la entrada A es una entrada muy pequeña, la entrada B es una entrada pequeña, pero de mayor tamaño que A, y C es la entrada de mayor tamaño.

Conjunto de datos	**A**	**B**	**C**
FIR (elementos señal/filtro)	1.000.000 / 99	10.000.000 / 99	10.000.000 / 999
K-means (observaciones)	204.800	494.020	819.200
BFS (nodos del grafo)	4.096	65.536	1.000.000

Tabla 9.6: Tamaño de los diferentes datos de entrada de las aplicacaiones.

Las implementaciones OmpSsCL, para los datos de entrada pequeños (A) son más lentas que las implementaciones realizadas, a mano, con OpenCL. Esto es debido, principalmente, al *overhead* introducido por el *runtime*. Sin embargo, observamos también

como para instancias más grandes, nuestra implementación automática es tan competitiva como la implementación manual usando OpenCL.

Una de las causas de la diferencia entre el rendimiento de OmpSsCL y OpenCL es la gestión de grupos de trabajo de OpenCL [5]. En el caso de la versión OmpSsCL ésta se gestiona de manera automática, lo que implica que en ciertas situaciones (datos de entrada pequeños) pueden no aprovecharse al máximo los recursos de los dispositivos.

La tabla 9.7 muestra los tiempos *elapsed* de ejecución correspondientes a la parte acelerada de las diferentes implementaciones y el *speedup* respecto a la versión secuencial ejecutada en una *CPU* para cada uno de los conjuntos de datos de entrada. Estos tiempos corresponden a la media de diez ejecuciones de cada aplicación.

Las filas etiquetadas como "secuencial" corresponden a una implementación de la aplicación ejecutada sobre una *CPU*, en las demás corresponden a ejecuciones en *GPU*.

Implementación	Conjunto de datos					
	A		**B**		**C**	
	Tiempo (s)	*SpeedUp*	Tiempo (s)	*SpeedUp*	Tiempo (s)	*SpeedUp*
FIR secuencial	0,138938	1x	1,380175	1x	12,784512	1x
FIR OmpSsCL	0,172360	0,80x	0,239905	5,75x	1,221338	10,47x
FIR OpenCL	0,017028	8,15x	0,220100	6,27x	1,255823	10,18x
K-means secuencial	1,222669	1x	6,285121	1x	5,835462	1x
K-means OmpSsCL	0,293704	4,16x	1,346238	4,66x	1,274166	4,58x
K-means OpenCL	0,245335	4,98x	1,148011	5,47x	0,999275	5,83x
BFS secuencial	0,000345	1x	0,006649	1x	0,169037	1x
BFS OmpSsCL	0,002715	0,13x	0,007195	0,92x	0,061713	2,73x
BFS OpenCL	0,001417	0,24x	0,006720	0,99x	0,080225	2,11x

Tabla 9.7: Tiempos de las diferentes aplicaciones e implementaciones de éstas, para los diferentes conjuntos de datos A, B y C.

9.6. Pruebas de ejecución concurrente

En esta sección mostraremos el comportamiento del sistema ejecutando tareas de manera concurrente en varios dispositivos al mismo tiempo, así como el control de dependencias efectuado en tiempo de ejecución cuando se trabaja con varios *threads*.

9.6.1. Ejecución concurrente en múltiples dispositivos

A continuación se analiza el comportamiento de una aplicación que lanza 20 tareas sin dependencias entre ellas.

Hasta el momento sólo se han mostrado trazas con un único hilo ya que únicamente utilizábamos un dispositivo. En este caso, vamos a usar tres dispositivos: una *GPU* y dos *CPU* (una por cada *core* de nuestro procesador). De modo que crearemos tres *threads*. Nótese que consideramos cada núcleo de la *CPU* como un dispositivo diferente. Esto es debido a que se ha utilizado la opción de fisión de dispositivos de OpenCL [5]. De otro modo, OpenCL consideraría todos los núcleos de la *CPU* como un único dispositivo.

La figura 9.20 muestra una traza de ejecución con tres filas (una por cada hilo de ejecución), mostrando en ella los dispositivos sobre los que se están ejecutando tareas resaltados con distintos colores.

La figura 9.21 muestra la correspondencia entre colores y dispositivos.

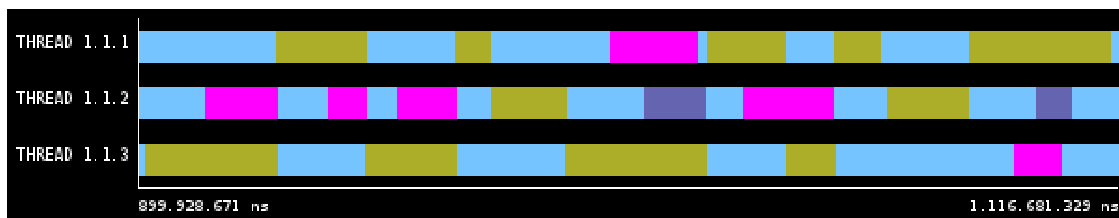

Figura 9.20: *Zoom* de la traza de ejecución mostrando la ejecución concurrente de tareas.

En la figura 9.20 se puede observar como se hace uso de los recursos heterogéneos del

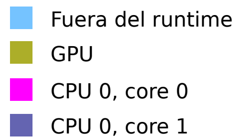

Figura 9.21: Leyenda de las trazas de ejecución para dispositivos.

sistema, ejecutando tareas en los distintos dispositivos del sistema de forma concurrente. Cabe destacar que en la última ejecución de tarea en el *thread* 1.1.1 (primera fila) corresponde a la ejecución de dos instancias de la tarea en el mismo dispositivo.

9.6.2. Control de dependencias

En esta sección se analiza el comportamiento del sistema cuando se intentan ejecutar diferentes tareas con dependencias entre ellas de manera concurrente.

La figura 9.22 muestra la traza de ejecución de una aplicación que lanza tres tareas: Una sobre *GPU* y las otras dos sobre *CPU*. La tareas que se ejecutan en *CPU* tienen dependencias de datos con la primera de ellas (ejecutada en la *GPU*).

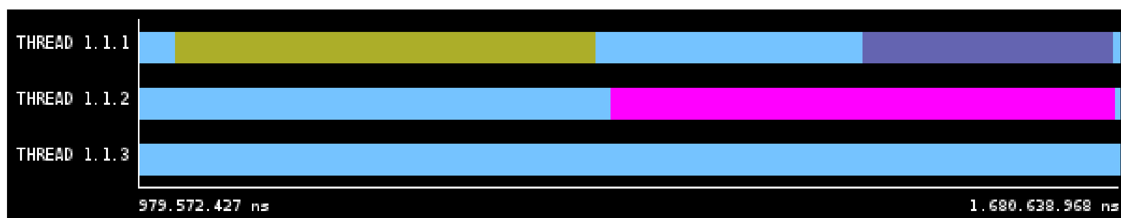

Figura 9.22: Zoom de una traza de ejecución mostrando la gestión de dependencias entre tareas.

En la figura podemos observar como la tarea de *GPU* correspondiente a la región marrón se ejecuta antes que las demás. Observamos también como las otras dos tareas ejecutadas en *CPU*, de colores rosa y morado, no comienzan su ejecución hasta que la tarea de la que dependen no ha finalizado.

Capítulo 10

Planificación y costes

En este capítulo se detalla el coste del proyecto así como la planificación de las tareas realizadas.

10.1. Costes del proyecto

Los costes del proyecto se han dividido en diferentes categorías: Recursos humanos y bienes amortizables.

Recursos humanos

En la tabla 10.1 se muestran en detalle los costes de personal necesarios para desarrollar el proyecto. Se han considerado tres roles distintos:

- **Coordinador**: Supervisa el proyecto y delimita las tareas para cumplir así con la planificación.
- **Analista**: Analiza los requisitos del proyecto, define el comportamiento del sistema y diseña su arquitectura.
- **Programador**: Realiza la implementación del proyecto con una tecnología concreta siguiendo la especificación del Analista.

Rol	Coste unitario	Dedicación	Coste total
Coordinador	60 € / hora	80 h	4800 €
Analista	40 € / hora	464 h	18560 €
Programador	35 € / hora	360 h	12600 €
Total			**35960 €**

Tabla 10.1: Coste de los recursos humanos del proyecto.

Bienes amortizables

En la tabla 10.2 se muestra el coste de los bienes amortizables, tales como equipo informático o mobiliario.

Descripción	Precio	Amortización	Coste unitario	Coste total
Tarjeta gráfica NVIDIA	105€	4 años	2,18 €/mes	21,8 €
Monitor de ordenador	142€	4 años	2,95 €/mes	29,5 €
Total				**51,3 €**

Tabla 10.2: Coste de los materiales amortizables del proyecto.

10.1.1. Resumen de costes

En la tabla 10.3 se muestra un resumen del coste del proyecto

Concepto	Coste
Recursos humanos	35960 €
Bienes amortizables	51,3 €
Total	**36011,3 €**

Tabla 10.3: Resumen de costes del proyecto.

10.2. Planificación

En la figura 10.1 se muestra un desglose de todas las tareas realizadas para completar el proyecto así como su planificación en forma de diagrama de gannt, mostrando para cada semana, qué tarea (o tareas) se realizan.

2011 2012

36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

Análisis Mercurium/OpenCL
Generacion de código
Generación de código CL
Transferencias de memoria
Gestón de dependencias
Instrumentación del runtime
Port aplicación de prueba
Análisis aplicación de prueba
Porting de tareas
Análisis de rendimiento
Optimizaciones de memoria
Implementación compilador
Implementacíón runtime
Validación
Análisis de rendimiento
Documentación
Preparación de la defensa

Figura 10.1: Planificación de las tareas

Capítulo 11

Conclusiones

A lo largo de este proyecto hemos desarrollado una infraestructura de compilación y ejecución (OmpSsCL) que facilita el desarrollo de aplicaciones sobre sistemas heterogéneos a través de una *API* basada en OmpSs, con el objetivo de aprovechar la gran capacidad de cálculo en paralelo de los aceleradores gráficos *GPU*.

En particular, se ha ampliado el *runtime* OpenCLNanos [6] para añadir soporte para la ejecución de tareas en paralelo utilizando la *API* del modelo de programación OpenCL, y se ha desarrollado la fase del compilador Mercurium para dar soporte a todas las construcciones y cláusulas utilizadas.

En el proyecto se ha trabajado con dispositivos de diferentes fabricantes (AMD y NVIDIA). Se ha observado que hay diferencias de rendimiento en OpenCL a pesar de que se trata de dispositivos con un rendimiento gráfico similar. Esto hace pensar que pueda tratarse de un bajo rendimiento del software, bien del controlador del dispositivo (*driver*) o bien de la implementación de su plataforma OpenCL, más que del propio hardware. También se han observado problemas de compatibilidad con los *kits* de desarrollo de distintos proveedores pese a tratarse del mismo estándar OpenCL.

Por otro lado, a través de los análisis de rendimiento se ha visto que un gran problema a la hora de trabajar con dispositivos aceleradores (concretamente *GPUs*) son las transferencias de memoria, siendo éstas, en la mayoría de los casos el principal cuello de botella.

En este proyecto tambien hemos hecho propuestas de extensiones al modelo de programación OmpSs con tal de poder reducir el numero de transferencias de memoria de una forma sencilla y eficiente desde el punto de vista del programador y el rendimiento.

El resultado ha sido una infraestructura de compilación que permite el uso de dispositivos heterogéneos de una forma eficiente a partir de un OmpSs extendido. El rendimiento obtenido con los códigos generados a partir de nuestro OmpSsCL son comparables al rendimiento obtenido realizando la codificación en OpenCL, de forma manual, pero con la ventaja de haber sido generados automáticamente.

Por otra parte, destacar también que durante el desarrollo del proyecto se han aplicado muchos de los conocimientos y habilidades adquiridas a lo largo de la carrera. En especial de la asignatura de *Compiladores*, que ha sido fundamental para poder comprender el funcionamiento de la infraestructura de compilación utilizada (Mercurium) y poder realizar así las numerosas transformaciones de código necesarias. También hacer mención a las asignaturas de *Arquitectura de Computadores* y *Programación Consciente de la Arquitectura*, que han aportado muchos de los conocimientos sobre las distintas arquitecturas usadas a lo largo del proyecto así como conocimientos sobre el análisis de rendimiento de aplicaciones.

Parte I

APÉNDICES

Apéndice A

Información del sistema

A.1. Resultados de la ejecución de CLInfo

CLInfo es una utilidad distribuida con el SDK de AMD que permite obtener información detallada sobre el entorno OpenCL de un sistema.

```
1  Number of platforms:                                  2
2    Platform Profile:                                   FULL_PROFILE
3    Platform Version:                                   OpenCL 1.1 CUDA
       4.2.1
4    Platform Name:                                      NVIDIA CUDA
5    Platform Vendor:                                    NVIDIA
      Corporation
6    Platform Extensions:
      cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing
      cl_nv_compiler_options cl_nv_device_attribute_query
      cl_nv_pragma_unroll
7    Platform Profile:                                   FULL_PROFILE
8    Platform Version:                                   OpenCL 1.1 AMD-
      APP-SDK-v2.4 (595.10)
9    Platform Name:                                      AMD Accelerated
       Parallel Processing
10   Platform Vendor:                                    Advanced Micro
      Devices, Inc.
11   Platform Extensions:                                cl_khr_icd
      cl_amd_event_callback cl_amd_offline_devices
```

```
  Platform Name:                                    NVIDIA CUDA
Number of devices:                                  1
  Device Type:
      CL_DEVICE_TYPE_GPU
  Device ID:                                        4318
  Max compute units:                                4
  Max work items dimensions:                        3
    Max work items[0]:                              1024
    Max work items[1]:                              1024
    Max work items[2]:                              64
  Max work group size:                              1024
  Preferred vector width char:                      1
  Preferred vector width short:                     1
  Preferred vector width int:                       1
  Preferred vector width long:                      1
  Preferred vector width float:                     1
  Preferred vector width double:                    1
  Native vector width char:                         1
  Native vector width short:                        1
  Native vector width int:                          1
  Native vector width long:                         1
  Native vector width float:                        1
  Native vector width double:                       1
  Max clock frequency:                              1800Mhz
  Address bits:                                     32
  Max memory allocation:                            268320768
  Image support:                                    Yes
  Max number of images read arguments:              128
  Max number of images write arguments:             8
  Max image 2D width:                               32768
  Max image 2D height:                              32768
  Max image 3D width:                               2048
  Max image 3D height:                              2048
  Max image 3D depth:                               2048
  Max samplers within kernel:                       16
  Max size of kernel argument:                      4352
  Alignment (bits) of base address:                 4096
  Minimum alignment (bytes) for any datatype:       128
  Single precision floating point capability
    Denorms:                                        Yes
    Quiet NaNs:                                     Yes
    Round to nearest even:                          Yes
```

```
55      Round to zero:                                        Yes
56      Round to +ve and infinity:                            Yes
57      IEEE754-2008 fused multiply-add:                      Yes
58    Cache type:                                             Read/Write
59    Cache line size:                                        128
60    Cache size:                                             65536
61    Global memory size:                                     1073283072
62    Constant buffer size:                                   65536
63    Max number of constant args:                            9
64    Local memory type:                                      Scratchpad
65    Local memory size:                                      49152
66    Kernel Preferred work group size multiple:              32
67    Error correction support:                               0
68    Unified memory for Host and Device:                     0
69    Profiling timer resolution:                             1000
70    Device endianess:                                       Little
71    Available:                                              Yes
72    Compiler available:                                     Yes
73    Execution capabilities:
74      Execute OpenCL kernels:                               Yes
75      Execute native function:                              No
76    Queue properties:
77      Out-of-Order:                                         Yes
78      Profiling :                                           Yes
79    Platform ID:                                            0x18e3fb0
80    Name:                                                   GeForce GTX 550
          Ti
81    Vendor:                                                 NVIDIA
        Corporation
82    Driver version:                                         295.33
83    Profile:                                                FULL_PROFILE
84    Version:                                                OpenCL 1.1 CUDA
85    Extensions:
        cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing
        cl_nv_compiler_options cl_nv_device_attribute_query
        cl_nv_pragma_unroll  cl_khr_global_int32_base_atomics
        cl_khr_global_int32_extended_atomics
        cl_khr_local_int32_base_atomics
        cl_khr_local_int32_extended_atomics cl_khr_fp64
86
87
88    Platform Name:                                          AMD Accelerated
          Parallel Processing
89  Number of devices:                                        1
```

```
90    Device Type:
          CL_DEVICE_TYPE_CPU
91    Device ID:                                          4098
92    Max compute units:                                  2
93    Max work items dimensions:                          3
94      Max work items[0]:                                1024
95      Max work items[1]:                                1024
96      Max work items[2]:                                1024
97    Max work group size:                                1024
98    Preferred vector width char:                        16
99    Preferred vector width short:                       8
100   Preferred vector width int:                         4
101   Preferred vector width long:                        2
102   Preferred vector width float:                       4
103   Preferred vector width double:                      0
104   Native vector width char:                           16
105   Native vector width short:                          8
106   Native vector width int:                            4
107   Native vector width long:                           2
108   Native vector width float:                          4
109   Native vector width double:                         0
110   Max clock frequency:                                2401Mhz
111   Address bits:                                       64
112   Max memory allocation:                              2147483648
113   Image support:                                      Yes
114   Max number of images read arguments:                128
115   Max number of images write arguments:               8
116   Max image 2D width:                                 8192
117   Max image 2D height:                                8192
118   Max image 3D width:                                 2048
119   Max image 3D height:                                2048
120   Max image 3D depth:                                 2048
121   Max samplers within kernel:                         16
122   Max size of kernel argument:                        4096
123   Alignment (bits) of base address:                   1024
124   Minimum alignment (bytes) for any datatype:         128
125   Single precision floating point capability
126     Denorms:                                          Yes
127     Quiet NaNs:                                       Yes
128     Round to nearest even:                            Yes
129     Round to zero:                                    Yes
130     Round to +ve and infinity:                        Yes
131     IEEE754-2008 fused multiply-add:                  No
132   Cache type:                                         Read/Write
```

```
133   Cache line size:                                  64
134   Cache size:                                       32768
135   Global memory size:                               4156063744
136   Constant buffer size:                             65536
137   Max number of constant args:                      8
138   Local memory type:                                Global
139   Local memory size:                                32768
140   Kernel Preferred work group size multiple:        1
141   Error correction support:                         0
142   Unified memory for Host and Device:               1
143   Profiling timer resolution:                       1
144   Device endianess:                                 Little
145   Available:                                        Yes
146   Compiler available:                               Yes
147   Execution capabilities:
148     Execute OpenCL kernels:                         Yes
149     Execute native function:                        Yes
150   Queue properties:
151     Out-of-Order:                                   No
152     Profiling :                                     Yes
153   Platform ID:                                      0x7f79a52e9800
154   Name:                                             Intel(R) Core(
        TM)2 CPU          6600  @ 2.40GHz
155   Vendor:                                           GenuineIntel
156   Driver version:                                   2.0
157   Profile:                                          FULL_PROFILE
158   Version:                                          OpenCL 1.1 AMD-
        APP-SDK-v2.4 (595.10)
159   Extensions:                                       cl_khr_fp64
        cl_amd_fp64 cl_khr_global_int32_base_atomics
        cl_khr_global_int32_extended_atomics
        cl_khr_local_int32_base_atomics
        cl_khr_local_int32_extended_atomics
        cl_khr_int64_base_atomics cl_khr_int64_extended_atomics
        cl_khr_byte_addressable_store cl_khr_gl_sharing
        cl_ext_device_fission cl_amd_device_attribute_query
        cl_amd_vec3 cl_amd_media_ops cl_amd_popcnt cl_amd_printf
```

Código A.1: Información del entorno OpenCL.

A.2. Contenido /proc/cpuinfo

El fichero /proc/cpuinfo de linux proporciona información detallada sobre los procesadores presentes en el sistema.

```
1  processor       : 0
2  vendor_id       : GenuineIntel
3  cpu family      : 6
4  model           : 15
5  model name      : Intel(R) Core(TM)2 CPU          6600  @ 2.40
     GHz
6  stepping        : 6
7  cpu MHz         : 2401.856
8  cache size      : 4096 KB
9  physical id     : 0
10 siblings        : 2
11 core id         : 0
12 cpu cores       : 2
13 apicid          : 0
14 initial apicid  : 0
15 fpu             : yes
16 fpu_exception   : yes
17 cpuid level     : 10
18 wp              : yes
19 flags           : fpu vme de pse tsc msr pae mce cx8 apic sep
     mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse
     sse2 ss ht tm pbe syscall nx lm constant_tsc arch_perfmon
     pebs bts rep_good nopl aperfmperf pni dtes64 monitor ds_cpl
     vmx est tm2 ssse3 cx16 xtpr pdcm lahf_lm dts tpr_shadow
20 bogomips        : 4803.71
21 clflush size    : 64
22 cache_alignment : 64
23 address sizes   : 36 bits physical, 48 bits virtual
24 power management:
25
26 processor       : 1
27 vendor_id       : GenuineIntel
28 cpu family      : 6
29 model           : 15
30 model name      : Intel(R) Core(TM)2 CPU          6600  @ 2.40
     GHz
31 stepping        : 6
32 cpu MHz         : 2401.856
```

```
33 cache size       : 4096 KB
34 physical id      : 0
35 siblings         : 2
36 core id          : 1
37 cpu cores        : 2
38 apicid           : 1
39 initial apicid   : 1
40 fpu              : yes
41 fpu_exception    : yes
42 cpuid level      : 10
43 wp               : yes
44 flags            : fpu vme de pse tsc msr pae mce cx8 apic sep
      mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse
      sse2 ss ht tm pbe syscall nx lm constant_tsc arch_perfmon
      pebs bts rep_good nopl aperfmperf pni dtes64 monitor ds_cpl
      vmx est tm2 ssse3 cx16 xtpr pdcm lahf_lm dts tpr_shadow
45 bogomips         : 4744.03
46 clflush size     : 64
47 cache_alignment  : 64
48 address sizes    : 36 bits physical, 48 bits virtual
49 power management:
```

Código A.2: Información del procesador.

Bibliografía

[1] *Official OpenMP Specification - Version 3.0*, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[2] Rosa M. Badia. Easy programming heterogeneous systems: Starss presentation and complexhpc, May 2011. http://www.cs.vu.nl/complexhpc2011/slides/complexHPC2011-StarSs.pdf.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, , and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE International Symposium on Workload Characterization (IISWC)*, October 2009.

[4] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a Proposal for Programming Heterogeneous Multi-core Architectures.

[5] Khronos Group. *Official OpenCL Specification - Version 1.2*, November 2011. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[6] Moisés Guillén Allés. Openmp to opencl: Aprovechamiento de los recursos heterogéneos del sistema. *UPCCommons*, 2011. http://upcommons.upc.edu/pfc/bitstream/2099.1/12473/1/69587.pdf.

[7] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[8] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, April 1965. ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf.