FACULTAD DE CIENCIAS

UNIVERSIDAD DE CANTABRIA Facultad de Ciencias

Proyecto Fin de Carrera

Equilibrio de Carga de Trabajo Dinámico en Sistemas Multi-CPU y Multi-GPU

Dynamic Load Balancing on Multi-CPU and Multi-GPU systems

Para acceder al Título de

INGENIERO EN INFORMÁTICA

Autor: Borja Pérez Pavón
Director: José Luis Bosque Orero

Junio 2014

Resumen

A causa del surgimiento de las GPUs como dispostivos de propósito general con gran capacidad de cómputo paralelo, los sistemas heterogéneos, que utilizan GPUs y CPUs, han cobrado especial protagonismo en el ámbito de la computación de alto rendimiento. A raíz de esto, se han desarrollado modelos de programación que permiten trabajar con dispositivos heterogéneos dentro de un sistema. Sin embargo, el soporte a multitud de dispositivos muy dispares, ofrecido por estos modelos, tiene un inconveniente fundamental: es necesario que la gestión del hardware se realice de forma individual para cada dispositivo, con las dificultades que esto conlleva. Asimismo, la obtención de un buen rendimiento, aprovechando todos los recursos del sistema, no resulta una tarea trivial, pues implica distribuir la carga de trabajo en función de la potencia de cómputo de cada dispositivo y gestionar la memoria adecuadamente, pues, en general, la memoria de cada dispositivo se encuentra separada.

El propósito de este proyecto es desarrollar un modelo de programación que permita distribuir el trabajo entre todos los recursos de los que disponga un sistema, de manera transparente al programador y aprovechando toda la potencia de cómputo disponible. Para ello, el modelo implementa 4 técnicas de balanceo de carga que permiten adaptar la forma de distribución de trabajo a las necesidades de cada aplicación, de modo que se aprovechen los recursos adecuadamente. Por otro lado, se introduce la noción de "sistema único", que permite que el usuario se comunique con el sistema completo, en lugar de con una miríada de dispositivos aislados. Esta idea permite que el código sea portable a otros sitemas con hardware diferente y, además, resulta en una disminución de la cantidad de las líneas de código necesarias para ejecutar una aplicación, facilitando la labor de programación.

Due to the emergence of GPUs as general purpose devices with great parallel computing capabilities, heterogeneous systems, which use GPUs and CPUs, have gained special prominence in the field of High Performance Computing. As a result of this, some programming models that make working with heterogeneous devices possible have been developed. However, the support of several different devices offered by these models has an important drawback: device management has to be done independently for each device, with all the difficulties that this carries. Obtaining of a good performance, making the most of all the available resources, is not a trivial task either, because it implies distributing the workload taking the computational power of each device into account and managing memory correctly, because, usually, the memory of the devices is separated.

The purpose of this project is developing a programming model that makes distributing the workload among all the resources available in a system possible, transparently to the programmer and making the most of the available computational power. To accomplish this, the model implements 4 load balancing techniques, so workload distribution can be adapted to the needs of each application and resources adequately used. The notion of using an only system, which enables the programmer to communicate with the whole system, instead of communicating with a myriad of isolated devices, is presented too. This idea makes source code portable to systems with different hardware and also reduces the source code length needed to run an application, making programming easier.

Índice general

Índice de figuras v

Índice de tablas vi

1. Introducción 1
- 1.1. GPUs y sistemas heterogéneos . . . 1
- 1.2. Equilibrio de carga de trabajo en sistemas heterogéneos . . . 4
- 1.3. Objetivos . . . 5
- 1.4. Plan de trabajo . . . 6
- 1.5. Estructura del documento . . . 7

2. Métodos y herramientas 8
- 2.1. OpenCL . . . 9
  - 2.1.1. Detección y gestión del hardware . . . 9
  - 2.1.2. Comunicación con los dispositivos y lanzamiento de kernels. El modelo de ejecución y programación . . . 11
  - 2.1.3. Modelo de memoria . . . 13
  - 2.1.4. Estructura general de un programa utilizando OpenCL para un sistema heterogéneo . . . 13
- 2.2. Métodos de equilibrio en sistemas heterogéneos . . . 15

3. Caracterización de *benchmarks* para sistemas heterogéneos 17
- 3.1. AMD APP SDK . . . 18
- 3.2. Rodinia . . . 18
- 3.3. Caracterización de Rodinia . . . 20
  - 3.3.1. Ejecuciones preliminares y primera selección . . . 20
  - 3.3.2. Instrumentalización de aplicaciones seleccionadas . . . 23
  - 3.3.3. Segunda selección . . . 24
- 3.4. El problema de reserva de recursos . . . 25

4. Librería de equilibrio de carga 27
- 4.1. Objetivos y arquitectura general del sistema . . . 27
- 4.2. Estructuras de datos . . . 28
  - 4.2.1. El super_contexto . . . 29
  - 4.2.2. El super_buffer . . . 29
  - 4.2.3. El super_kernel . . . 29
- 4.3. Arquitectura a nivel usuario . . . 30
  - 4.3.1. Gestión del entorno de trabajo . . . 30

4.3.2. Gestión de memoria 31
4.3.3. Gestión de *kernels* 31
4.3.4. Liberación de recursos 32
4.4. Estructura general de un programa utilizando Maat 32
4.5. Lanzamiento de super_kernels y balanceo de carga 34
4.5.1. Balanceo estático 35
4.5.2. Balanceo dinámico 36
4.5.3. Balanceo guiado (*guided*) 37
4.5.4. Balanceo guiado heterogéneo 37

5. Validación experimental **39**
5.1. Entorno de experimentación 39
5.2. Resultados lavaMD 43
5.2.1. Sistema MultiCPU-MultiGPU 43
5.2.2. Sistema MultiCPU 46
5.2.3. Sistema MultiGPU 47
5.3. Resultados RAP 48
5.3.1. Sistema MultiCPU-MultiGPU 48
5.3.2. Sistema MultiCPU 50
5.3.3. Sistema MultiGPU 51

6. Conclusiones **52**
6.1. Implementación del equilibrio de carga 52
6.2. Permitir el desarrollo de código portable 54
6.3. Simplificación de la labor de programación 54
6.4. Trabajo futuro 55

Bibliografía **56**

Índice de figuras

1.1. Arquitectura de una GPU 3

2.1. Modelo *host-device* 10
2.2. Problema de ejemplo: Suma paralela de vectores 12
2.3. Modelo de memoria de OpenCL 14

3.1. Aplicaciones del grupo 2 22
3.2. Aplicaciones del grupo 3 22
3.3. Aplicaciones del grupo 4 23
3.4. Aplicaciones del grupo 4 desglosadas 24

4.1. Comportamiento de las técnicas de balanceo 38

5.1. *Baseline* 41
5.2. Comparativa del tiempo total en LavaMD 43
5.3. Comparativa del tiempo de *kernel* en LavaMD 45
5.4. Comparativa del tiempo de comunicación en LavaMD 45
5.5. Comparativa del tiempo de *kernel* en LavaMD con una CPU 47
5.6. Comparativa del tiempo de *kernel* en LavaMD con 2 GPUs 47
5.7. Comparativa del tiempo total en RAP 48
5.8. Comparativa del tiempo de comunicación en RAP 49
5.9. Comparativa del tiempo de *kernel* en RAP con una CPU 50
5.10. Comparativa del tiempo de *kernel* en RAP con 2 GPUs 51

Índice de tablas

3.1. Aplicaciones de Rodinia 19
3.2. Clasificación de las aplicaciones 21

4.1. Comparativa de longitud de código 33
4.2. Relación entre funciones Maat y funciones OpenCL 34

5.1. Potencias normalizadas y *speedup* teórico por aplicación 41
5.2. Parámetros de balanceo por aplicación 42
5.3. Speedups LavaMD 44
5.4. Speedups RAP 49

Capítulo 1

Introducción

1.1. GPUs y sistemas heterogéneos

Tradicionalmente, los sistemas de computación han estado formados, independientemente de su escala, por dispositivos idénticos. Es decir, los sistemas eran homogéneos. Sin embargo, de un tiempo a esta parte, los sistemas han ido eliminando esta homogeneidad en virtud de la versatilidad que ofrece disponer de dispositivos de características diferentes. Así, la presencia de dispositivos específicos, diseñados para optimizar ciertas aplicaciones (aquellas con un elevado grado de paralelismo) permite mejorar tanto el rendimiento como el consumo, al hacer posible ajustarse a las necesidades de cada aplicación concreta. El desarrollo de los sistemas heterogéneos ha cobrado aún más fuerza con el surgimiento y rápida evolución de las GPUs, como dispositivos capaces de realizar tareas de cómputo de propósito general, mejorando el rendimiento y trabajando de forma más eficiente que las CPUs para ciertas aplicaciones. De esta manera, a día de hoy, la mayoría de los sistemas listados en el Top 500 [1] son heterogéneos en mayor o menor medida.

Las GPUs son dispositivos de cómputo cuya función original era encargarse de tareas relacionadas con la aceleración de las operaciones de procesamiento gráfico. Por este motivo, las GPUs han sido siempre dispositivos muy vinculados a la industria del videojuego. El resultado de esta vinculación es que, debido a la gran cantidad de recursos que se invierten en los gráficos por computador, las GPUs hayan evolucionado más rápidamente que las CPUs tradicionales. Sin embargo, este propósito tan específico también hace que una GPU difiera bastante, en lo que a terminología y arquitectura se refiere, de una CPU convencional. Originalmente, las GPUs estaban formadas por un conjunto de procesadores heterogéneos, teniendo cada uno de ellos como propósito resolver una tarea específica del procesamiento gráfico. De hecho, como el procesamiento gráfico

sólo implica operaciones con enteros, al trabajar con píxeles en una imagen discreta, las GPUs ni tan siquiera disponían de soporte para coma flotante. Con el paso del tiempo, pese a mantener las características diferenciadoras que las hacen destacar en el procesamiento paralelo, las GPUs han ido pareciéndose más a las CPUs multinúcleo, al construirse utilizando procesadores homogéneos de propósito general para obtener una mayor flexibilidad de programación.

Las GPUs, de acuerdo a la taxonomía de Flynn [2], son dispositivos SIMD. Esto es, trabajan con vectores de datos, aplicando la misma instrucción a cada una de las posiciones de este. Trabajar con vectores de datos tiene una consecuencia directa: el ratio entre acceso a memoria y cómputo es muy grande. Para ocultar esta elevada latencia en el acceso a memoria, las GPUs recurren al *multithreading* masivo de grano fino. Esta es la principal diferencia entre GPUs y CPUs pues estas, para ocultar la latencia, se valen de complejas jerarquías de memoria (aunque, en el acercamiento entre GPUs y CPUs, las primeras han empezado a incorporar caches en sus modelos más actuales). Para dar soporte a este grado de *multithreading*, la GPU se construye internamente como un conjunto de multiprocesadores con soporte *multithread*, que en la nomenclatura de Nvidia se denominan ***stream multiprocessors***. A su vez, cada uno de estos multiprocesadores está formado por un conjunto de unidades de coma flotante de simple y doble precisión y de procesamiento de enteros, denominadas ***stream processors***. La diferencia fundamental entre una CPU de varios núcleos y un *stream multiprocessors* es que, el segundo, está formado por muchos más núcleos que el primero, los cuales, por contra, son mucho más simples. Así, una CPU tradicional confía en pocos núcleos potentes, con ejecución fuera de orden y gran capacidad de predicción de salto por ejemplo, mientras que la GPU cuenta con una gran cantidad de *stream processors* simples. Esta elevada cantidad de *stream processors* también implica que las GPUs tienen bancos de registros mucho más grandes que las CPUs. La figura 1.1 contiene la arquitectura general de una GPU. En definitiva y de manera simplificada, una GPU es un procesador vectorial multinúcleo y *multithread* de grano fino, formado por un conjunto de *stream multiprocessors*, que a su vez están integrados por *stream processors*. Así, por ejemplo, una GPU actual dispone de 15 *stream multiprocessors*, cada uno con 192 *stream processors*, pudiendo ejecutar hasta 2048 *threads* por multiprocesador. Cabe destacar que, por sus orígenes como acelerador al servicio de una CPU, las memorias de las CPUs y las GPUs son independientes y se encuentran separadas.

Con todo esto, la gran cantidad de *stream processors* que forman las GPUs hacen que estos dispositivos sean ideales para ejecutar tareas masivamente paralelas. Sin embargo, estas tareas deben ser relativamente simples, pues los saltos condicionales, en general, castigan mucho el rendimiento de las GPUs. Esto ocurre porque es necesario que ambos resultados del salto se ejecuten secuencialmente, de forma que algunos de los recursos

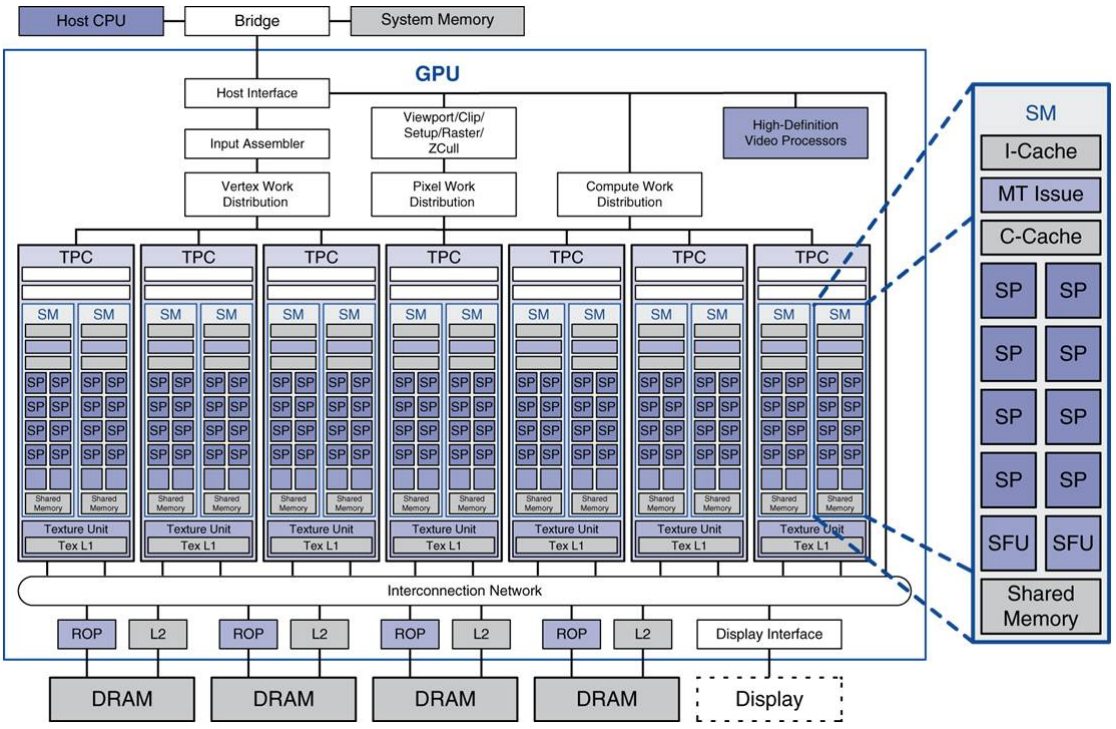

FIGURA 1.1: Arquitectura de una GPU

de la GPU tienen que esperar y el rendimiento no es óptimo. Por su parte, las CPUs, al tener menos núcleos, se enfrentan peor a problemas con un elevado grado de paralelismo, pero están mejor preparadas para resolver problemas complejos por poseer núcleos más potentes.

La gran capacidad de cómputo, la posibilidad de aprovechar el paralelismo de datos inherente al trabajo en forma vectorial y la eficiencia energética (una GPU es capaz de obtener un rendimiento de 4,29 TFlops/s con un consumo de 255 W), motivaron la incursión de las GPUs en el cómputo de propósito general. Sin embargo, las diferencias de arquitectura y filosofía que generan este interés en las GPUs, también son causa de un inconveniente: es necesario desarrollar un modelo de programación adecuado para las necesidades de estos dispositivos. Así surgieron CUDA[3] y OpenCL[4], modelos de programación que permiten considerar a las GPUs (y a una gran variedad de dispositivos de otro tipo en el caso de OpenCL) como coprocesadores al servicio de una CPU, la cual les envía las tareas que deben ejecutar y los datos que precisen. Sin embargo, aún queda una pregunta por responder: ¿Cómo aunar lo mejor de ambos mundos, aprovechando toda la potencia de cómputo que ofrecen los sistemas heterogéneos?

1.2. Equilibrio de carga de trabajo en sistemas heterogéneos

El principal problema del que adolecen los modelos de programación disponibles para GPUs radica en su concepción: el considerar a dichos dispositivos como coprocesadores al servicio de una CPU. Este hecho hace que las GPUs (o cualquiera de los dispositivos considerados como coprocesadores) se comporten como unidades independientes, las cuales no tienen ningún conocimiento de las tareas que están realizando otros coprocesadores (ni de su existencia siquiera). En definitiva, si se desea aprovechar toda la potencia de cómputo de un sistema heterogéneo, la responsabilidad de distribuir la carga de trabajo entre los dispositivos disponibles recae por entero en manos del programador. Esta labor no resulta para nada trivial, pues implica conocer y gestionar los dispositivos del sistema, distribuir el trabajo y gestionar las estructuras de memoria de los dispositivos entre otras tareas. Además, a los aspectos intrínsecos al reparto de trabajo, se une que considerar los dispositivos como coprocesadores dificulta la portabilidad del código. Esto es, una modificación en el sistema en lo que al número de coprocesadores se refiere resultará, en general, en una modificación del código si desean aprovecharse todos los recursos disponibles. El resultado de esta situación es que, en muchos casos, las aplicaciones que se lanzan en entornos heterogéneos terminan siendo ejecutadas teniendo sólo en cuenta el dispositivo de mayor potencia, quedándose el resto a la espera mientras este realiza el trabajo y, probablemente, desperdiciando potencia de cómputo. En definitiva, en estos casos, la carga de trabajo se encuentra desequilibrada, realizando unos dispositivos la mayor parte del trabajo y el resto poco o nada. Esto es, en general se utiliza un modelo síncrono en el que la CPU envía trabajo a la GPU y se queda a la espera, en lugar de colaborar en ese trabajo.

Para tratar de resolver el problema de que algunos recursos del sistema se encuentren sobrecargados frente a otros que están infrautilizados, se han desarrollado técnicas o algoritmos de equilibrio de carga de trabajo que tratan de aprovechar el potencial paralelo del que disponen los sistemas heterogéneos. Las técnicas propuestas para tratar de reducir esta infrautilización de recursos pueden dividirse en dos líneas diferenciadas: el aprovechamiento del paralelismo funcional y el del paralelismo de datos.

El aprovechamiento del **paralelismo funcional** consiste en ejecutar tareas diferentes, paralelamente, en los diversos dispositivos del sistema. Es decir, el paralelismo funcional busca una ganancia de rendimiento a través de la ejecución de tareas independientes. Existe gran cantidad de artículos que exploran esta opción, como por ejemplo [5] o [6]. El principal problema de esta técnica es que, si lo que se desea es mejorar el rendimiento de una única aplicación, es necesario dividirla en tareas que no presenten dependencias entre sí y, en general, las tareas en las que puede dividirse un problema no son suficientes

para aprovechar todos los recursos de sistemas con GPUs. Así, el paralelismo funcional se considera una forma de aprovechar el paralelismo de grano grueso.

Por su parte, el **paralelismo de datos** explota la ejecución, en diferentes dispositivos, de una misma tarea sobre datos de entrada distintos. Esto es, mientras que en el caso funcional se explotaba el paralelismo a través de la independencia de tareas, en el paralelismo de datos se aprovecha la independencia de datos para una sola tarea. Así, el equilibrio de carga se obtiene distribuyendo los datos de entrada entre los dispositivos disponibles de acuerdo a su capacidad de cómputo, de modo que todos se encuentren ocupados la mayor parte de tiempo posible. Este problema es más adecuado al estilo de trabajo de la GPU, pero mucho más complejo, debido a la gestión de memoria. El estudio que se ha realizado de este tipo de paralelismo, sobre todo en sistemas que aunan GPUs y CPUs, es mucho más limitado. Por este motivo y por las posibilidades que ofrece, este proyecto se centra en la implementación del equilibrio de carga mediante paralelismo de datos.

En definitiva, el objetivo de todas estas técnicas es conseguir que la carga se encuentre equilibrada en el sistema. En general, diremos que la carga de un sistema está equilibrada cuando el trabajo se distribuye entre los dispositivos disponibles de acuerdo a su potencia de cómputo, de forma que estos estén ocupados todo el tiempo. En general, las técnicas de balanceo de carga pueden agruparse en dos categorías: el balanceo **estático** y el **dinámico**. La diferencia entre estos métodos radica en el tamaño y número de las porciones en las que se divide el trabajo, así como en momento en el que se realiza la asignación de estas porciones a los dispositivos. En el primero, el trabajo se divide en tantas porciones como dispositivos tenga el sistema, con lo que estas serán moderadamente grandes, asignándosele una porción a cada dispositivo. En este balanceo, la asignación se realiza una sola vez al comienzo de la ejecución y no se modifica hasta el final. Por su parte, en el balanceo dinámico, el trabajo se divide en porciones muy pequeñas (muchas más que dispositivos), de forma que se le van asignando partes de trabajo a los dispositivos según estos van quedándose libres. Por estos motivos, el balanceo estático es simple y minimiza la sobrecarga por sincronización entre GPU y CPU, pero se adapta mal a cargas de trabajo variables. El balanceo dinámico tiene las propiedades opuestas: es más complejo y representa una mayor sobrecarga de sincronización, pero se adapta mejor a cargas heterogéneas.

1.3. Objetivos

Considerando las secciones anteriores, este proyecto tiene como propósito extender el modelo de programación de OpenCL para cumplir los siguientes objetivos:

- Implementar el equilibrio de carga de trabajo sobre sistemas heterogéneos, ofreciendo diversas técnicas de balanceo y aprovechando toda la potencia de cómputo disponible.
- Permitir el desarrollo de código portable. Esto es, código que pueda ser ejecutado en cualquier sistema aprovechando sus capacidades heterogéneas sin necesidad de realizar modificaciones. Para conseguir esto, se presenta la noción de sistema como una unidad con la que el programador se comunica.
- Simplificar la labor de programación, aportando una visión del sistema como una unidad en lugar de como una CPU con un conjunto de coprocesadores asociados. De esta manera, el programador enviará el trabajo al sistema completo, sin preocuparse de la naturaleza y cantidad de los dispositivos de los que este disponga.

1.4. Plan de trabajo

La metodología seguida para cumplir los objetivos comentados en el apartado anterior es la siguiente:

1. Realizar un estudio previo del tema del equilibrio de carga a partir de la bibliografía disponible. De esta manera, se obtiene una perspectiva del trabajo realizado y de los problemas existentes en el reparto de trabajo.
2. Estudiar OpenCL como herramienta a partir de la cual se desarrollará la librería que este proyecto tiene como objetivo.
3. Analizar los conjuntos de *benchmarks* disponibles y caracterizar sus aplicaciones, con objeto de escoger aquellas más apropiadas para valorar los resultados de rendimiento obtenidos utilizando la librería desarrollada.
4. Paralelizar manualmente algunas aplicaciones en un sistema heterogéneo, para familiarizarse con las dificultades asociadas al reparto de trabajo en un sistema heterogéneo y obtener una perspectiva de las necesidades de OpenCL de cara a implementar el equilibrio de carga.
5. Desarrollar la librería que implementa el equilibrio de carga y cumple los objetivos listados en la sección anterior. Esta ha sido la tarea que ha supuesto una mayor carga de trabajo de las que conforman el proyecto.
6. Sintonizar experimentalmente los parámetros necesarios para la librería.

7. Realizar, utilizando las aplicaciones escogidas, un análisis del rendimiento que producen las diversas técnicas de equilibrio de carga implementadas en la librería, en comparación con unos resultados base.

8. Analizar los resultados experimentales y obtener conclusiones.

1.5. Estructura del documento

El presente documento se ha estructurado en una serie de capítulos, que describen el trabajo realizado en este proyecto. Estos capítulos son:

El capítulo 2 se centra en introducir OpenCL como herramienta utilizada para implementar la librería de balanceo de carga. Este capítulo es fundamental para entender el funcionamiento de la librería, al basarse esta en formato y funciones en OpenCL estándar. Por otro lado, también se ofrece una visión general del estado del arte en el balanceo de carga en la bibliografía.

El capítulo 3 contiene un análisis de algunos de los conjuntos de *benchmarks* más importantes de los utilizados en el ámbito científico. Asimismo, se presenta una caracterización del las aplicaciones de la *suite* Rodinia y se escogen las aplicaciones más adecuadas para realizar las pruebas de rendimiento necesarias para comprobar el cumplimiento de los objetivos de la librería. Además, se introducen las métricas seleccionadas para realizar el posterior análisis de rendimiento.

En el capítulo 4 se presenta Maat, la librería desarrollada para implementar el equilibrio de carga cumpliendo los objetivos antes mencionados. En este capítulo se explican las estructuras de datos utilizadas, así como la arquitectura de la librería a nivel usuario. Además, también se explican detalladamente las cuatro técnicas de distribución de carga implementadas.

El objetivo del capítulo 5 es presentar los resultados experimentales extraídos de las pruebas de rendimiento de la librería, para cada una de las técnicas de equilibrio de carga, aplicándolas además a diversos sistemas. Adicionalmente, también se ofrece una descripción del entorno de ejecución y de las condiciones en las que se han realizado las pruebas de rendimiento.

Finalmente, el capítulo 6 presenta las conclusiones del proyecto y el trabajo futuro. Asimismo, se valora el cumplimiento de los objetivos planteados en este capítulo.

Capítulo 2

Métodos y herramientas

El aprovechamiento de todos los recursos ofrecidos por un sistema heterogéneo, por sus particularidades y dificultades intrínsecas, no es una tarea nada trivial. El primer problema a abordar es la necesaria cooperación entre dispositivos hardware que serán, no sólo diferentes sino, potencialmente, de diferente arquitectura. Este hecho motiva que, en aras de posibilitar la portabilidad del código y facilitar la programación, se desarrolle el modelo de programación paralela OpenCL para la programación de sistemas heterogéneos.

Por otro lado, dejando de lado los detalles directamente ligados con la tecnología, el balanceo de carga, per se, es un tema que, por su complejidad, suscita interés investigador. No en vano, recordemos que nos encontramos ante un problema que se encuadra en la clase de complejidad de los NP-Duros. Además, dadas las condiciones especiales de nuestra situación, el reparto de trabajo debe realizarse de forma rápida pues, en caso contrario, el *speedup* obtenido con la distribución del trabajo entre los diferentes dispositivos se perdería en el proceso mismo de elección del reparto. Por todo esto, el balanceo de carga en sistemas heterogéneos es un tema en evolución, con constantes aportaciones nuevas en la bibliografía más referenciada en este campo de investigación, que será preciso estudiar antes de comenzar con una aproximación propia al problema.

El propósito de este capítulo es introducir la librería OpenCL como herramienta que se utilizará a la hora de abordar el problema del balanceo carga, así como presentar, con sus fortalezas y debilidades, el estado del arte en el balanceo carga en los sistemas de computación heterogéneos

2.1. OpenCL

OpenCL (Open Computing Language) [4] es un *framework*, desarrollado y mantenido por el Khronos Group, cuyo objetivo fundamental es facilitar la programación heterogénea. Uno de los pilares clave de OpenCL es su elevada portabilidad, siendo posible la ejecución del mismo código en dispositivos de muy variada índole. De este modo, el código puede ejecutarse indistintamente tanto en una CPU como en una GPU (independientemente de su fabricante), así como en una FPGA o, en general, en cualquier clase de hardware que implemente el estándar OpenCL y para el cual exista el *driver* apropiado.

Para soportar esta disparidad de dispositivos, OpenCL se presenta como un estándar formado por dos piezas fundamentales:

- Una API, la cual ofrece las funciones necesarias para la gestión del entorno de ejecución paralela (dispositivos, *buffers* de memoria, etc)
- Un lenguaje de programación en el cual se codificarán los *kernels*, que son las unidades algorítmicas de OpenCL, esto es, las partes de programa que se ejecutarán paralelamente en los dispositivos.

De este modo, teniendo en cuenta las características del estándar, el modelo de programación de OpenCL se basa en las nociones de ***Host*** y ***Device***. Así, un dispositivo (una CPU) actúa como *host* y se encarga de la gestión del entorno de trabajo, mientras que el resto actúan como *device* y se encargan de ejecutar el trabajo, en forma de *kernels*, encomendado por el primero. En definitiva, los *device* se comportan como co-procesadores del *host*, ejecutando *kernels*. Estos ***kernels*** son la unidad algorítmica de OpenCL y son, símplemente, funciones (habitualmente cortas y simples), escritas en el lenguaje OpenCL, el cual es una extensión de C, que serán ejecutadas de forma paralela en el dispositivo en el que sean lanzadas. Por otro lado, el *host* y los *devices*, tienen memorias separadas, lo cual hace necesaria una comunicación entre ellos con objeto de transmitir los parámetros necesarios para el cómputo y los resultados de este. En vista de esta forma de trabajo, hay una serie de aspectos del estándar OpenCL que cabe estudiar. A saber:La detección y gestión del hardware disponible, la comunicación con el hardware y el lanzamiento de *kernels* y el modelo de memoria.

2.1.1. Detección y gestión del hardware

Con objeto de desacoplar el hardware y el código, lo cual es básico para la portabilidad, el estándar define la noción de **plataforma**, que es una implementación de la API de

OpenCL específica de un vendedor. De esta manera, es el vendedor el encargado de desarrollar *driver* que cumpla con el estándar. A su vez y para favorecer la escalabilidad, se define el **dispositivo** (device), dentro de una plataforma, como un conjunto de **unidades de cómputo**. Así, por ejemplo, una CPU es tratada como un conjunto de unidades de cómputo (los núcleos que la conforman) pudiendo, a decisión del programador y de forma lógica, realizar un tratamiento de estas unidades como dispositivos de pleno de derecho. De esta manera, si así se desea, el reparto de trabajo (*kernels*) puede realizarse con una granularidad de unidad de cómputo o, lo que es lo mismo, pueden repartirse tareas a los núcleos individuales de una CPU. A su vez, las unidades de cómputo están formadas por componentes más pequeños, los **elementos de proceso**, sin embargo el estándar no permite controlar estos elementos directamente. Así, si nos fijamos en la estructura de una GPU, OpenCL la tratará como un conjunto de unidades de cómputo (los *stream multiprocessors*) cada cual formado por una conjunto de elementos de proceso (los *stream processors*).

Con todo esto, en el caso más general, un sistema estará formado por tantas plataformas como fabricantes diferentes estén presentes, cada una con sus dispositivos asociados, los cuales, a su vez, están integrados por las unidades de cómputo que los conforman, integradas por un conjunto de elementos de proceso. La API ofrece las funciones necesarias para la detección, inicialización y posterior liberación de dispositivos y plataformas, así como para la división de un dispositivo en las unidades de cómputo que lo conforman para su uso separado. La figura 2.1 presenta el modelo *host-device* así como los elementos que forman un dispositivo.

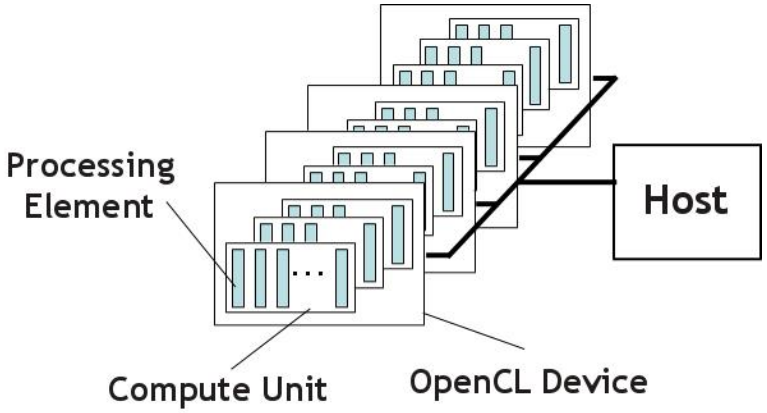

FIGURA 2.1: Modelo *host-device*

2.1.2. Comunicación con los dispositivos y lanzamiento de kernels. El modelo de ejecución y programación

La estructura básica para la gestión de los dispositivos dentro de OpenCL es el **contexto**. Esta estructura está conformada, símplemente, por un conjunto de dispositivos que necesariamente deben pertenecer a la misma plataforma. Al ser el elemento básico de gestión, el contexto, de un modo u otro, está presente en todas las operaciones de comunicación con los dispositivo, incluyendo la transmisión de datos de *host* a dispositivo y viceversa. Esta transmisión de información se realiza utilizando los ***buffers***, que son una estructura propia de OpenCL la cual, conceptualmente, representa un puntero a memoria que es válido en todos los dispositivos pertenecientes a un contexto. Una vez creado, el *buffer* puede ser leído o escrito utilizando las funciones correspondientes de la API. Así, de forma efectiva, un *buffer* y sus datos pueden ser compartidos entre todos los dispositivos de un contexto, pero no entre dispositivos de contextos diferentes. Este detalle, que puede pasar inadvertido en una primera lectura, es un aspecto a tener muy en cuenta a la hora de aprovechar el potencial de la programación heterogénea. Esto se debe a que, considerando que dispositivos de plataformas diferentes no pueden pertenecer a un mismo contexto y que sólo pueden compartirse *buffers* dentro de un contexto, la compartición de datos entre dispositivos de plataformas diferentes no está soportada por el estándar y es una tarea que queda totalmente bajo la responsabilidad del programador.

El contexto es también la base para la creación de las **colas de comandos**, que son el elemento del estándar a utilizar cuando se desea que un dispositivo tome alguna acción, como puede ser la escritura en un *buffer* de memoria o el lanzamiento de un *kernel*. La ejecución paralela de los *kernels* está gobernada por la noción de **elemento de trabajo** (*work-item*). De esta manera, el número de *work-items* es una cantidad que debe ser fijada por el programador de forma explícita en el lanzamiento de un *kernel* y que establece el número de veces que este debe ejecutarse. Así, cada elemento de trabajo puede conocer su posición dentro del conjunto de elementos y trabajar únicamente sobre los datos que le corresponden. El número de *work-items* o ***global_work_size*** se define como un rango de una, dos o tres dimensiones, de forma que pueda elegirse la opción que mejor se adapte a las características del problema que va a resolverse.

A modo ilustrativo, pensemos en la suma de dos vectores (que llamaremos A y B) dejando el resultado en un tercer vector (C). En este caso, una opción de paralelización, aunque no la única, es el definir un kernel que realice la suma del contenido de la posición i de los vectores A y B y deje el resultado en la misma posición de C (Figura 2.2). En este caso, esta posición i estaría gobernada por la identificación del elemento de trabajo dentro del conjunto, de forma que lanzaríamos tantos elementos de trabajo como posiciones tengan

los vectores a sumar, definidos a lo largo de una única dimensión. Por contra, si la tarea es la aplicación de desenfoque gaussiano a una imagen, lo lógico es definir los elementos a lo largo de dos dimensiones pues una imagen, por lo general, se trata como una matriz. En cualquier caso y como puede apreciarse, el modo en el que se definan los work-items es altamente dependiente de la lógica seguida a la hora de codificar el kernel que va a ejecutarse.

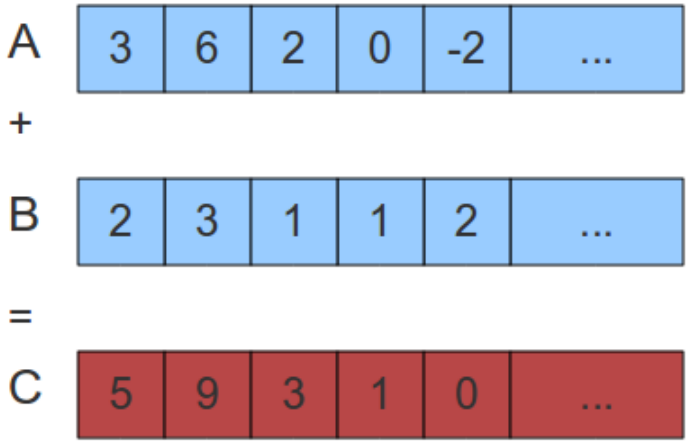

FIGURA 2.2: Problema de ejemplo: Suma paralela de vectores

Adicionalmente, el estándar define **grupos de trabajo** (*workgroups*), que son conjuntos de *work-items* que comparten memoria y que pueden realizar operaciones de sincronización entre sí. Cabe destacar que el número de grupos de trabajo se define indirectamente, pues lo que se especifica en el lanzamiento del kernel, junto con el *global_work_size*, es el tamaño que debe tener cada uno de los grupos de trabajo, llamado ***local_work_size***, el cual debe ser divisor del *global_work_size*. Teniendo todo esto en cuenta, el trabajo dentro de un *kernel*, por lo general, está gobernado por dos índices que permiten identificar unívocamente a los elementos de trabajo. Estos son el **id local** y el **id global** que identifican respectivamente al elemento de trabajo dentro su grupo de trabajo y dentro todo el conjunto de elementos de trabajo que se definieron con el *global_work_size*. Con todo esto, salta a la vista que el *local_work_size* y el *global_work_size* son elementos fundamentales para la corrección y el rendimiento de los *kernels*. Finalmente, junto con estos dos elementos, el estándar también permite definir el ***global_work_offset***, que especifica un desplazamiento a utilizar en el cálculo del id global en los elementos de trabajo.

2.1.3. Modelo de memoria

Existe otro factor importante dentro de OpenCL que debe ser considerado: La gestión de memoria. La causa de que este aspecto del estándar sea reseñable es, una vez más, la diversidad de dispositivos (que previsiblemente presentarán sistemas de memoria muy dispares) que pueden implementar el estándar OpenCL. Por poner un ejemplo, la mayoría de las CPUs modernas soportan memorias cache mientras que, en general, las GPUs no lo hacen (aunque cabe destacar que las más modernas sí que incluyen memorias cache e incluso permiten configurar su tamaño). Para afrontar esta situación, el estándar propone un sistema de memoria lógico e independientes del hardware, siendo el fabricante concreto el responsable de realizar el mapeo como resulte oportuno para obtener los mejores resultados de rendimiento de su hardware. Este modelo de memoria abstracto define cuatro espacios de memoria:

- **Memoria global** la cual es visible para todas las unidades de cómputo de un dispositivo. En general, es equivalente a la memoria principal. La latencia de acceso a esta memoria es muy elevada.
- **Memoria de constantes**. Está reservada para datos que van a ser accedidos de forma simultánea o valores que no van a cambiar. Suele ubicarse en una parte de la memoria principal.
- **Memoria local**, la cual es privada a una unidad de cómputo y compartida dentro de un grupo de trabajo. En una CPU, por norma, equivale a la memoria cache.
- **Memoria privada**, la cual es privada a un elemento de trabajo. Suele mapear sobre el banco de registros.

La figura 2.3 contiene una representación gráfica del modelo de memoria de OpenCL con sus diversos espacios.

2.1.4. Estructura general de un programa utilizando OpenCL para un sistema heterogéneo

Teniendo todo lo anterior en cuenta, la forma de trabajo utilizando el estándar OpenCL resulta bastante inmediata. A grandes rasgos, esta puede estructurarse en los siguientes pasos:

- Detección de las plataformas del sistema.
- Detección e inicialización de los dispositivos de las plataformas deseadas.

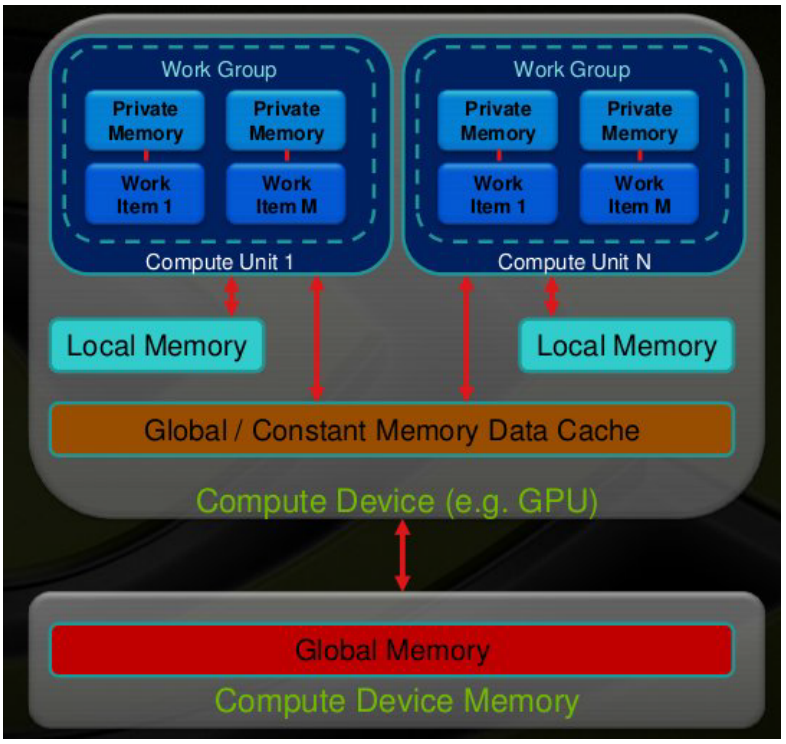

FIGURA 2.3: Modelo de memoria de OpenCL

- Subdivisión de los dispositivos, agrupando algunas de sus unidades de cómputo, para tratarlas como dispositivos independientes (Opcional).
- Creación de tantos contextos como requieran las plataformas utilizadas, que contengan a los dispositivos con los que se desea trabajar.
- Creación de las colas de comandos que resulten necesarias para posibilitar la comunicación con cada uno de los dispositivos.
- Creación de los *kernels* necesarios como estructuras de datos que representa el trabajo a ejecutar paralelamente pos los dispositivos(Teniendo en cuenta que no pueden compartirse entre plataformas).
- Creación de los *buffers* de memoria necesarios para que los *kernels* puedan operar y copia de datos (Teniendo en cuenta que los *buffers* no pueden compartirse entre plataformas).
- Asociación de parámetros a los *kernels*
- Lanzamiento de los *kernels*.
- Lectura de resultados una vez se ha completado la ejecución de los *kernels*. Los resultados se encontrarán distribuidos entre todos los dispositivos.
- Liberación de cada uno de los *buffers*, *kernels*, dispositivos y contextos.

2.2. Métodos de equilibrio en sistemas heterogéneos

Por su complejidad e interés, se han publicado gran cantidad de artículos relacionados con el reparto del trabajo entre los recursos disponibles en un sistema. Sin embargo, pese a este esfuerzo, el problema aún no ha sido satisfactoriamente resuelto. En esta sección se explorarán algunas de estas alternativas, diferenciando entre las técnicas de balanceo que se introdujeron en la sección 1.2.

En primer lugar, dentro de la opción del balanceo estático, en [7] se propone un entorno de trabajo que permite trabajar con múltiples GPUs como si fueran un único dispositivo. Además, también se propone un método para distribuir los dato necesarios para la ejecución (y, de forma efectiva, la carga) entre las GPUs de forma óptima. Esto es, se distribuyen los datos de modo que el tiempo de transferencia sea mínimo y un dispositivo tenga datos contiguos, de forma que se evite el lanzamiento innecesario de más kernels. Para conseguir esto, es preciso realizar un análisis de acceso a *buffers* en tiempo de ejecución. Sin embargo, esta aproximación sufre de una debilidad fundamental: El entorno de trabajo propuesto sólo considera las GPUs y, además, asume que todas son idénticas, asignándole la misma cantidad de trabajo a cada una. En definitiva, esta aproximación no tiene en cuenta la posible heterogeneidad entre GPUs.

Por su parte, en [8] se propone una librería, que encapsula las llamadas a la API de OpenCL, para el balanceo de carga estático entre todos los dispositivos presentes en un sistema. Esta aproximación, permite obtener un buen rendimiento, pero carece de la versatilidad que ofrece un método de balanceo dinámico.

Por otro lado, en la versión dinámica, [9] propone un método de balanceo de carga entre todos los dispositivos presentes en un sistema (ya sean CPUs o GPUs). El método propuesto se basa en el cálculo del rendimiento relativo de los dispositivos a partir de la ejecución de la primera porción de trabajo para, a continuación, repartir el resto del trabajo teniendo en cuenta los rendimientos relativos de la primera parte. A la luz de las pruebas experimentales realizadas, este método de balanceo de carga permite obtener un buen rendimiento, sin embargo, en el artículo se pasa por alto la posibilidad de que la primera porción de trabajo no sea representativa de la carga de trabajo real del problema como conjunto, realizándose en tal caso una distribución de carga errónea utilizando los rendimientos relativos obtenidos con la primera porción. Como dato adicional, el método propuesto en este artículo se ha probado experimentalmente utilizando únicamente *kernels* modificados para aprovechar el balanceo de carga y no los *kernels* estándar de una *suite* de *benchmarks*.

En [10] se propone una librería centrada en el balanceo de carga dinámico entre diversos dispositivos. Sin embargo, no considera el caso en el que la carga debe ser distribuida

entre dispositivos heterogéneos (GPUs y CPUs). De hecho, la aplicación de las técnicas propuestas a entornos heterogéneos queda como trabajo futuro en el propio texto. Adicionalmente, si se desea realizar un balanceo de carga correcto, las técnicas y tecnologías utilizadas en este artículo imponen la necesidad de realizar diversas implementaciones de cada algoritmo a ejecutar paralelamente, en función del entorno en el que el algoritmo en cuestión vaya a ejecutarse (CPUs, una GPU o varias GPUs). Esta necesidad complica la labor de programación y reduce drásticamente la portabilidad del código desarrollado.

Por último, existe una tercera aproximación al problema del balanceo de carga: el reparto de trabajo con entrenamiento. Siguiendo esta línea, [11] propone una librería de balanceo de carga con una base de datos que almacena la información de los tiempos de ejecución de las aplicaciones. Así, ante la ejecución de una aplicación desconocida, se asigna la misma cantidad de trabajo a la CPU y a la GPU y se almacene el tiempo de respuesta en la base de datos. Tras esto, las sucesivas ejecuciones de la misma aplicación, realizarán el reparto de trabajo en base a los datos disponibles y a unas fórmulas matemáticas propuestas, y almacenarán el nuevo tiempo de respuesta en la base de datos. Así, una vez pasado un número indeterminado de ejecuciones, la base de datos contendrá información que permitirá realizar un reparto de trabajo cercano al óptimo. La principal deficiencia de esta estrategia, como es obvio, es que sólo es útil en sistemas en los que se ejecutan frecuentemente las misma aplicaciones, pero no en aquellos en los que no se cumpla esta premisa, pues el rendimiento obtenido para aplicaciones nuevas es en general malo. Además, el artículo sólo considera la situación en la que el sistema dispone de una CPU y una GPU, sin estudiar el problema del balanceo de carga en aquellos sistemas que dispongan de más de una GPU y/o CPU. Por otra parte, el acceso a una base de daros, cuando el tamaño de esta es grande, introduce una sobrecarga considerable en comparación con la ganancia obtenida al aplicar técnicas de este tipo.

Capítulo 3

Caracterización de *benchmarks* para sistemas heterogéneos

Uno de los aspectos clave a considerar en un proyecto de estas características es la forma de evaluar el grado de mejora obtenido con la solución propuesta. Para esto, resulta interesante utilizar herramientas estándar, bien establecidas y documentadas; escogiendo siempre de entre todas las disponibles aquella o aquellas que mejor se ajusten a las necesidades concretas del problema.

En el campo de la computación paralela, existen diversas *suites* (conjuntos) de *benchmarks* que gozan de un amplio reconocimiento. Entre ellas, pueden destacarse: PARSEC, desarrollado por Intel y la universidad de Princeton; SPLASH-2, propuesto por la Universidad de Berkeley; Rodinia, propuesto por la Universidad de Virginia y el AMD APP SDK (AMD Accelerated Parallel Processing technology Software Development Kit). Cada uno de estos *benchmarks* tiene particularidades que los hacen especialmente apropiados para analizar el rendimiento de sistemas concretos.

Tanto PARSEC como SPLASH-2 se centran en aplicaciones con cargas de trabajo multithread y sistemas homogéneos de memoria compartida. Esto es, multiprocesadores on-chip (CMPs). Rodinia y el SDK de AMD, por su parte, están enfocados en sistemas heterogéneos. Además las dos primeras *suites* están programadas, de acuerdo con los dispositivos en los que se centran, utilizando pthreads u OpenMP, mientras que las dos segundas utilizan, entre otros, OpenCL. Por estos motivos, Rodinia y el AMD APP SDK son las *suites* ideales para el escenario de este proyecto. En las siguientes secciones se estudiarán Rodinia y el SDK de AMD y se caracterizarán las aplicaciones que componen la primera de estas *suites*, con objeto de seleccionar aquellas que más se ajusten a las necesidades de este proyecto. Finalmente, ante el comportamiento totalmente homogéneo en lo que a carga de trabajo se refiere de las aplicaciones de las *suites* de *benchmarks*, al

final de este capítulo se introduce una aplicación que presenta carga de trabajo variable, factor que es imprescindible al experimentar con una librería de balanceo de carga.

3.1. AMD APP SDK

El AMD APP SDK [12] es una plataforma de desarrollo completa, que tiene como objetivo el desarrollo de aplicaciones aceleradas utilizando diversas tecnologías implementadas por AMD. Entre estas, se encuentra la programación para sistemas heterogéneos, con CPUs de varios núcleos, GPUs y *Accelerated Processing Units* (APUs). El SDK contiene gran cantidad de aplicaciones implementadas utilizando OpenCL, así como algunos *benchmarks*. Sin embargo, estas características, que convertirían el SDK en la opción ideal para realizar pruebas de rendimiento, se ven ensombrecidas por la simplicidad (en general) de las aplicaciones que ofrece. De esta manera, las aplicaciones, incluso con los tamaños más grandes admitidos, presentan unos tiempos de respuesta demasiado pequeños para constituir una prueba de rendimiento de interés. Por su parte, los *benchmarks*, por centrarse en áreas demasiado específicas del rendimiento (Mediciones de ancho de banda, optimizaciones de memoria...), tampoco son útiles para las pruebas que son deseables en el escenario de este proyecto. Por este motivo, una vez analizadas, estas aplicaciones han sido desechadas.

3.2. Rodinia

Propuesto por la Universidad de Virginia en 2009, la *suite* de *benchmarks* Rodinia [13] surge de la necesidad de evaluar el rendimiento de sistemas formados por CPUs de varios núcleos y aceleradores de diverso tipo. La popularidad de estos sistemas se encontraba (y encuentra) en alza y, hasta ese momento, no existía ninguna herramienta de *benchmarking* apropiada, pues todas se centraban símplemente en las CPUs, ignorando completamente las GPUs.

Rodinia, a diferencia del resto de *suites*, se diseñó desde sus inicios con los sistemas heterogéneos en mente, aprovechando por ejemplo jerarquías de memoria diferentes de las tradicionales (*scratchpad*, unidades de texturas...). Otro de los elementos diferenciadores de Rodinia con respecto al resto de conjuntos de *benchmarks*, es que ofrece una diversa selección de aplicaciones y *kernels*. Estas abarcan dominios tan variados como la minería de datos, las simulaciones físicas o el procesado de imágenes con fines médicos, entre otros. De este modo, los sistemas pueden evaluarse bajo patrones de comportamiento muy variados, hecho que resulta de especial interés. En la tabla 3.1 se presenta

una síntesis de las aplicaciones que componen la *suite*, indicando su nombre, función y parámetros de entrada.

Nombre	Función	Parámetros de entrada
B+Tree	Búsqueda en árboles b+, algoritmo ampliamente usado por sistemas gestores de bases de datos en sus consultas	Dos ficheros, uno especificando los parámetros j y k de la búsqueda y otro con los datos
Back Propagation	Reconocimiento de patrones. Implementa un algoritmo de aprendizaje máquina para el entrenamiento de una red neuronal por capas	El tamaño de la red a entrenar
Breadth-First Search	Búsqueda en anchura en un grafo	Fichero conteniendo el grafo
Computational Fluid Dynamics Solver (CFD Solver)	Dinámica de fluidos. Resuelve las ecuaciones de Euler de tres dimensiones para fluidos compresibles	Un fichero con las características del fluido
Gaussian Elimination	Resuelve sistemas de ecuaciones lineales aplicando el método de Gauss	Un fichero con la matriz que representa el sistema
Heartwall Tracking	Procesamiento de imágenes médicas. Realiza seguimiento a los cambios de forma de las paredes del corazón de un ratón a partir de una secuencia de imágenes	Un fichero de vídeo y el número de *frames* a procesar
Hotspot	Realiza simulaciones de la temperatura estimada de un procesador basándose en su "*floor plan*" arquitectural y medidas de consumos	Un fichero de temperaturas, un fichero de consumos y tres parámetros propios del algoritmo
K-Means	Minería de datos. Implementa el conocido algoritmo de mismo nombre	Un fichero con los elementos a agrupar
LavaMD	Dinámica molecular. Calcula el potencial y recolocación de las partículas debido a las fuerzas mutuas existentes entre ellas en un espacio tridimensional	Un entero especificando la granularidad con la que debe dividirse el espacio
Leukocyte Tracking	Procesado de imágenes médicas. Realiza seguimiento a leucocitos en imágenes de vídeo de vasos sanguíneos	Un fichero de vídeo y el número de *frames* a procesar
LU Decomposition	Resuelve sistemas de ecuaciones lineales mediante el método de álgebra lineal homónimo	La matriz especificando el sistema de ecuaciones
Myocite	Modela y simula el comportamiento de los miocitos (células musculares cardíacas) y su actividad eléctrica	El número de segundos a simular
Nearest Neighbor	Minería de datos. Realiza la búsqueda de los k vecinos más cercanos de entre un conjunto de datos sin estructura. En concreto, utiliza datos de huracanes	El fichero con los nombres de los ficheros a utilizar, el número de registros a retornar, la latitud y la longitud a utilizar
Needleman-Wunsch	Implementa el método de mismo nombre para la optimización del alineamiento de secuencias de ADN	La longitud de las secuencias y un valor de penalización
Particle Filter	Estima la localización de un objeto en movimiento a partir de medidas con ruido y de su posición y una idea de su trayectoria dentro de la teoría bayesiana	Las dimensiones x e y del frame, el número de partículas y el número de *frames a procesar*
Pathfinder	Halla el camino de menor peso acumulado desde un punto de origen hasta un punto de destino en una cuadrícula utilizando programación dinámica	Las dimensiones x e y de la cuadrícula el tamaño de la pirámide
SRAD	Elimina ruido de imágenes sin perder datos importantes. Fundamentalmente, se utiliza en el procesado de imágenes ultrasónicas y de radar	Las dimensiones de la imagen, el número de iteraciones y el coeficiente de saturación
Stream Cluster	Resuelve el problema del "clustering online"	Seis parámetros propios del algoritmo

Tabla 3.1: Aplicaciones de Rodinia

3.3. Caracterización de Rodinia

Como puede apreciarse, Rodinia está formado por un conjunto de aplicaciones de muy diferentes dominios y, previsiblemente, comportamientos. Por esta causa, es fundamental realizar una caracterización detallada de las aplicaciones, de modo que se disponga de una noción del comportamiento de cada aplicación en función de sus parámetros de entrada, así como del rendimiento que cada uno de los componentes del sistema puede proporcionar. Una vez obtenida esta información, resultará sencillo escoger aquellas aplicaciones más adecuadas para comprobar el rendimiento de la librería de balanceo de carga que se propone en este proyecto.

Para la caracterización de los *benchmarks* se ha seguido una metodología jerárquica articulada alrededor de las siguientes etapas:

1. Realización de una serie de ejecuciones preliminares con diversos tamaños de problema, calculando la media de los tiempos de ejecución para cada tamaño. Primera selección.
2. Instrumentalización de aquellas aplicaciones que presentan interés por su comportamiento y que lo precisen (algunas aplicaciones ya están adecuadamente instrumentalizadas).
3. Ejecuciones de las aplicaciones instrumentalizadas y selección de la aplicación o aplicaciones que, por su comportamiento, resultan de interés para realizar las pruebas de rendimiento de la librería.

3.3.1. Ejecuciones preliminares y primera selección

El primer paso de la caracterización es realizar una ejecución de cada uno de los *benchmarks* con diversos tamaños de problema. El objetivo de esta fase es doble: En primer lugar, establecer una relación entre los parámetros de entrada de las aplicaciones y su comportamiento, pues la evolución del tiempo de ejecución de cada aplicación está gobernada por uno o varios de los parámetro de entrada que aparecen en la tabla 3.1, los cuales es necesario conocer.

En segundo lugar, realizar una primera caracterización de grano grueso de las aplicaciones en función de su tiempo ejecución en relación con el tamaño de problema, así como de la porción del tiempo de ejecución que corresponde a tiempo de kernel. Esto resulta importante pues uno de los principales problemas de las *suites* de *benchmarks* es la simplicidad de las aplicaciones que los forman, resultando esto en tiempos de ejecución

pequeños, que hacen que las aplicaciones no resulten de utilidad en un análisis experimental. Por otro lado, es deseable que las aplicaciones seleccionadas tengan un tiempo de *kernel* que constituya un porcentaje representativo del tiempo total, pues, como demuestra la Ley de Amdahl [14], el rendimiento de una aplicación paralela está limitado por su parte secuencial. Aplicado a este escenario, no tendría sentido realizar pruebas de rendimiento para una librería de balanceo de carga con aplicaciones en las que la mayor parte del tiempo de ejecución no sea paralelizable por representar tiempo de comunicación entre CPU y GPU. La realización de este paso sirve, en definitiva, como primer filtro para focalizar la atención en aquellas aplicaciones con una complejidad suficiente.

Considerando los resultados de estas ejecuciones preliminares y la metodología de caracterización ya explicada, existen 4 grandes grupos de aplicaciones:

1. Aquellas que, por sus características, no resultan fácilmente caracterizables al no ser posible realizar una estimación de la complejidad del problema y establecer una relación entre parámetros de entrada y tiempo de respuesta.

2. Las que muestran un tiempo de respuesta pequeño o que no varía significativamente al aumentar el tamaño del problema. Los resultados de las aplicaciones de este grupo pueden consultarse en la figura 3.1.

3. Las que presentan una porción de tiempo de kernel pequeña en relación con el tiempo de respuesta. La figura 3.2 contiene los resultados para estas aplicaciones.

4. Aquellas en las que el tiempo de ejecución es suficientemente grande, variando este además en relación con alguno de los parámetros de entrada, y en las que el tiempo de kernel representa una porción considerable del tiempo de ejecución. Los tiempos medidos para las aplicaciones de este grupo están en la figura 3.3.

Criterio de agrupación	Aplicaciones
Difícilmente caracterizables	Hotspot, nearest neighbor, streamcluster
Tiempo de ejecución pequeño o no variable con los parámetros de entrada	B+tree[1], backprop[2] , gaussian[3]
Porción de tiempo de kernel pequeña	Bfs, lud, leukocyte, srad[4], pathfinder, nw
Comportamiento adecuado	Heartwall, myocite, particleFilter, lavaMD

Tabla 3.2: Clasificación de las aplicaciones

[1]También podría encuadrarse en el grupo 3
[2]También podría encuadrarse en el grupo 3
[3]Aunque parece comportarse adecuadamente respecto del tiempo, el *benchmark* no permite utilizar un tamaño de problema mayor que 2048, el cual no genera un tiempo de ejecución suficientemente grande
[4]También podría encuadrarse en el grupo 2

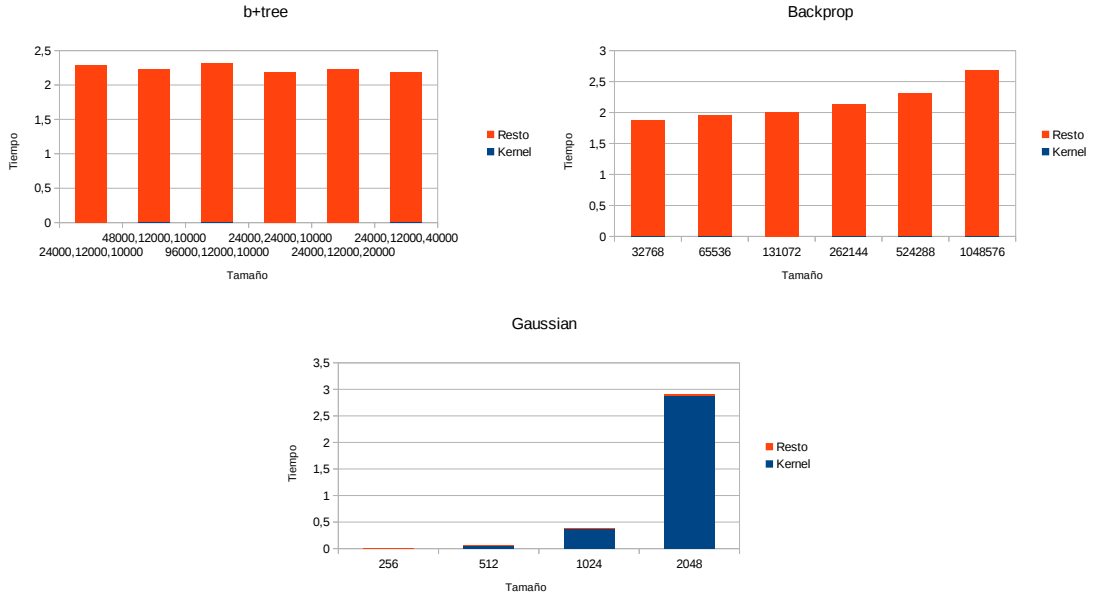

FIGURA 3.1: Aplicaciones del grupo 2

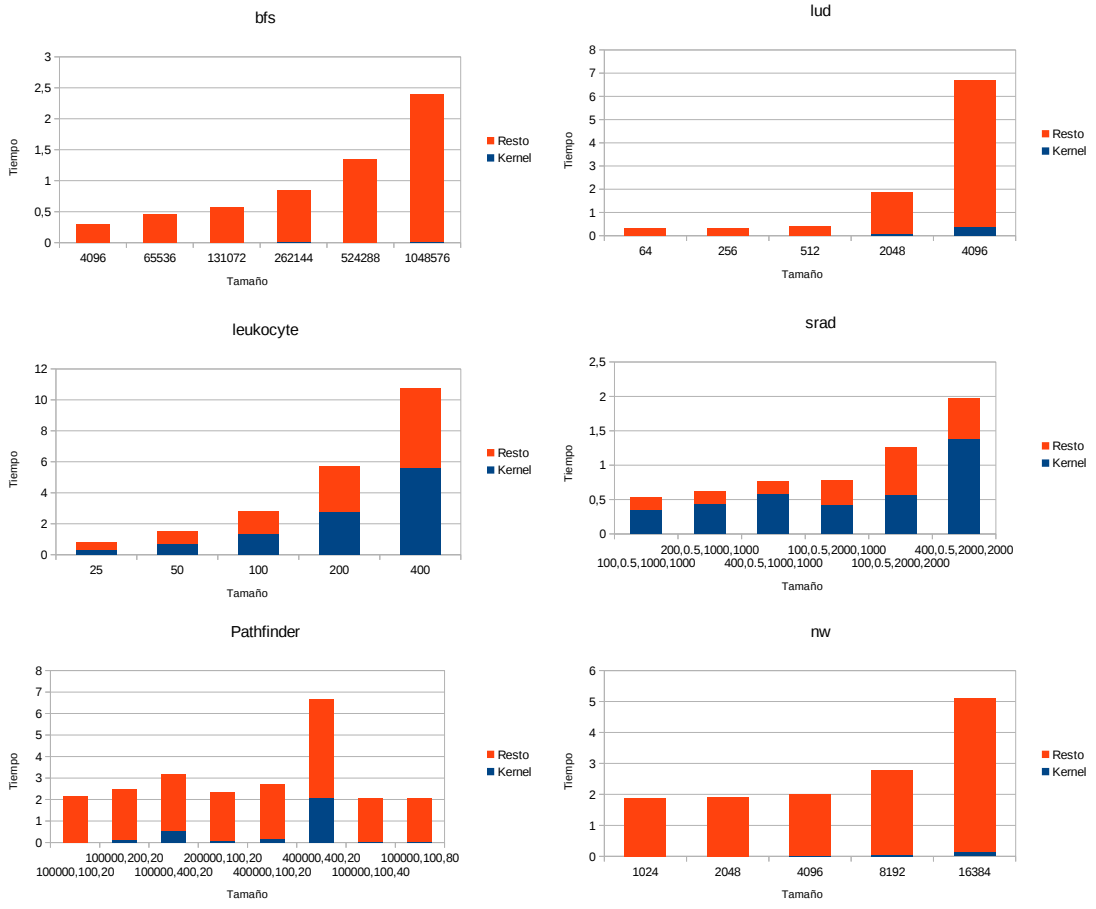

FIGURA 3.2: Aplicaciones del grupo 3

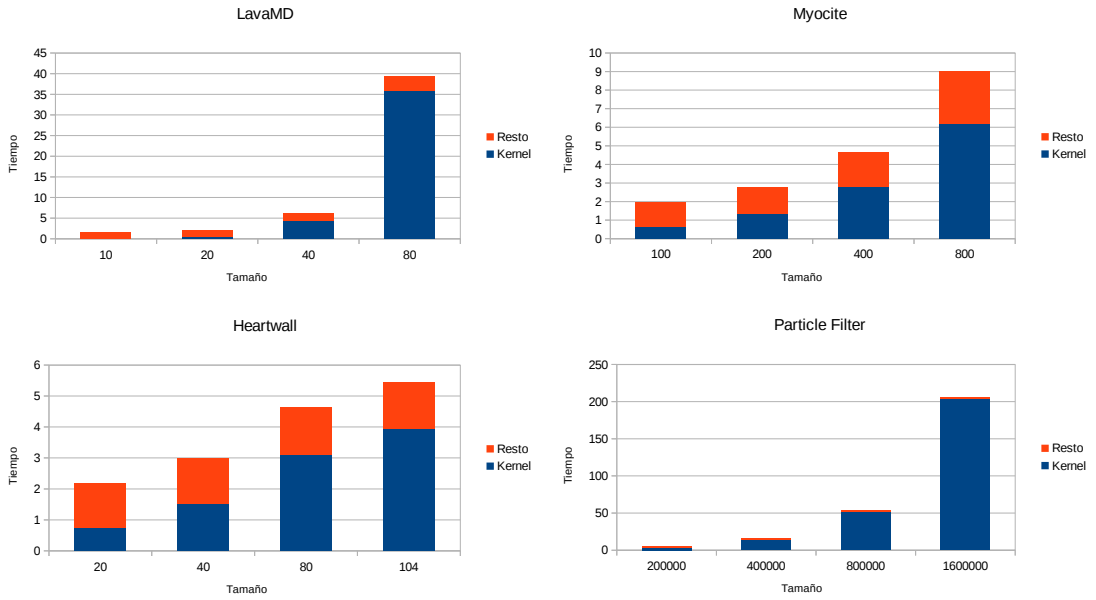

FIGURA 3.3: Aplicaciones del grupo 4

Teniendo en cuenta lo introducido al inicio de esta subsección, las aplicaciones de los tres primeros grupos, por sus características, no resultan interesantes para las pruebas experimentales de este proyecto, con lo que pueden ser descartadas sólo con este análisis preliminar de grano grueso. Por su parte, las aplicaciones del cuarto grupo parecen mostrar un comportamiento adecuado a las necesidades del proyecto. Por este motivo, estas aplicaciones deben ser instrumentalizadas con objeto de realizar un análisis más pormenorizado de su comportamiento.

3.3.2. Instrumentalización de aplicaciones seleccionadas

Una vez realizada la primera selección, es necesario obtener una información maś exacta del comportamiento de cada una de las aplicaciones escogidas. Para ello, es preciso instrumentalizar los *benchmarks*, esto es, modificar su código de modo que se obtengan datos relativos al rendimiento. En este caso, la instrumentalización tiene por objetivo obtener información precisa de los tiempos de ejecución, detallados por etapa, de cada aplicación. Esto permitirá realizar una selección de grano más fino de las aplicaciones a utilizar en las pruebas de rendimiento de la librería, así como, una vez llegada la fase de experimentación, analizar fácilmente el impacto sobre cada una de las fases de ejecución de la utilización de la librería de balanceo de carga propuesta. Para conseguir esto, se ha modificado el código de los *benchmarks* convenientemente, de forma que muestren información detallada de su tiempo de ejecución. Se ha prestado especial atención, por ser representativas para el proyecto, a las etapas de inicialización del driver, copia de

parámetros a la GPU, ejecución del kernel propiamente, copia de resultados a la CPU y liberación de recursos.

3.3.3. Segunda selección

Tras instrumentalizar el código, el último paso es realizar nuevas ejecuciones de los *benchmarks* seleccionados con diversos tamaños de problema. Estas ejecuciones producirán la información necesaria para realizar una nueva selección de aplicaciones, más precisa que la anterior. A la hora de realizar esta segunda selección de las aplicaciones, se ha tomado como criterio la evolución de los tiempos de comunicación entre CPU-GPU frente al tamaño del problema. La justificación de tomar la evolución del tiempo de comunicación como criterio, es que este parámetro es el factor que más limita el tiempo de ejecución de las GPUs (por norma, el cuello de botella se encuentra siempre en las comunicaciones entre CPU y GPU). Además, muchos artículos omiten este tiempo de comunicación, mostrando los resultados sólo en función del tiempo de kernel. Esto desvirtúa los resultados, pues la comunicación es un elemento que siempre va estar presente en el cómputo (de hecho, como se ha argumentado, es un factor limitante) y no puede obviarse.

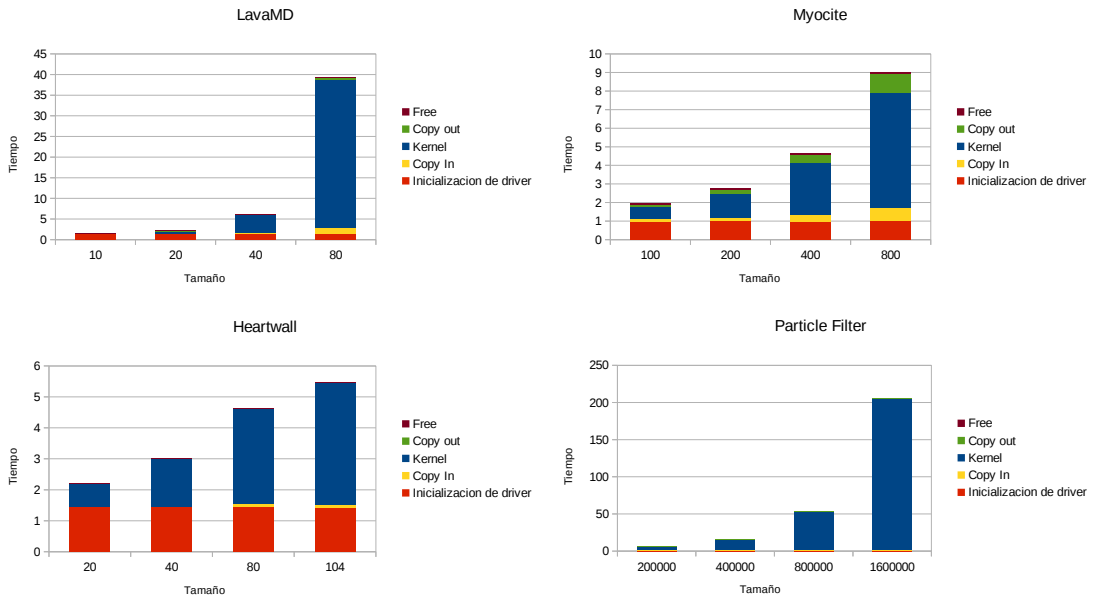

FIGURA 3.4: Aplicaciones del grupo 4 desglosadas

Como puede observarse en la figura 3.4, existe una diferencia fundamental en el comportamiento de las aplicaciones seleccionadas en lo que respecta al tiempo relativo a la comunicación. Por un lado, el tiempo de comunicación, en LavaMD y Myocite, es creciente y constituye una porción significativa del tiempo de cómputo. Por otro lado, en Heartwall y Particle Filter, este tiempo es insignificante y apenas variable. Teniendo

en cuenta que el tiempo de comunicación que, en última instancia, está ligado con la cantidad de memoria que maneja la aplicación, es un factor limitante del rendimiento, resulta interesante seleccionar aplicaciones en las que este tiempo crezca con el tamaño del problema y, además, resulte una parte representativa del tiempo de respuesta total. Por otro lado, los *benchmarks* de rodinia están en general pensados para ser ejecutados sobre una sola GPU, no siendo demasiado relevante el tiempo de comunicación para la suite. Sin embargo, al escalar el sistema y utilizar varias GPUs y CPUs, el volumen de datos manejado debe ser significativo, como lo sería en condiciones reales al fin y al cabo. Teniendo estos criterios criterios en cuenta, las aplicaciones seleccionadas para el posterior análisis son: Myocite y LavaMD.

Una vez llegados a este punto, cabe analizar el funcionamiento de las propias aplicaciones para realizar un último cribado que permita escoger aquellas que sean idóneas para las pruebas de rendimiento. Así, por su construcción y propósito, myocite, que es una aplicación de análisis de vídeo frame a frame (con una ejecución de *kernel* por *frame*), presenta dependencias de datos entre *frames* que dificultan la paralelización de la aplicación utilizando varios dispositivos. De hecho, la propia estructura del kernel de esta aplicación hace que no se puedan respetar las dependencias de datos si se desea ganar rendimiento ejecutando la aplicación en varios dispositivos. Para obtener una mejora de rendimiento, sería preciso codificar un kernel específico para este propósito, pero esto va en contra de la filosofía de realizar pruebas con aplicaciones estándar, no programadas específicamente.

De este modo y teniendo en cuenta lo argumentado a lo largo de toda esta sección, la aplicación ideal para la realización del estudio del rendimiento es LavaMD.

3.4. El problema de reserva de recursos

Pese a la cantidad de aplicaciones que contiene, Rodinia tiene una carencia fundamental, que es que todos sus *benchmarks* presentan una carga de trabajo homogénea. Esto es, dos porciones de trabajo cuales quiera, de mismo tamaño, representarán una carga de trabajo también igual (o muy similar). La carga de trabajo en estas aplicaciones se encuentra, en definitiva, equilibrada. Sin embargo, no todas las las aplicaciones se comportan de este modo. Por tanto, teniendo además en cuenta el objetivo de la librería, que es el balanceo de carga en sistemas heterogéneos, tiene sentido realizar pruebas con alguna aplicación que presente carga también heterogénea. El problema elegido para ello es el problema de reserva de recursos (*Resource Allocation Problem o RAP*) [15]. Este es un problema de programación dinámica ampliamente conocido, consistente en la asignación de recursos a tareas cumpliendo algún tipo de condición. Los problemas de este tipo, por definición

presentan una carga desbalanceada. La implementación del algoritmo para resolver el problema RAP es la propuesta en [10].

Capítulo 4

Librería de equilibrio de carga

4.1. Objetivos y arquitectura general del sistema

Tal y como se ha expuesto en el capítulo 2, OpenCL ofrece las funcionalidades necesarias para hacer posible el reparto de trabajo entre todos los dispositivos disponibles en un sistema heterogéneo. Sin embargo, pese a ofrecer los medios precisos para implementarlo, el balanceo de carga en OpenCL no está libre de dificultades. Uno de los principales inconvenientes es que el estándar, por su construcción, no facilita el que se realice una gestión de los dispositivos portable. Esto es, el estándar proporciona las funciones necesarias para trabajar con varios dispositivos, pero no dispone de la capacidad de gestionar el sistema y sus recursos como un todo al cual se le puede enviar trabajo, que es, al fin y al cabo, como se comportan los computadores tradicionalmente. Por contra, cada dispositivo debe ser gestionado individualmente, siendo responsabilidad del programador el decidir qué tareas y en qué momento deben ser desempeñadas por cada dispositivo. En definitiva, el mantenimiento del entorno de trabajo heterogéneo, con todas las dificultades que conlleva, queda en manos del programador. Esta situación resulta en que, potencialmente, cualquier cambio en el hardware implique la necesidad de modificar el código de las aplicaciones, de forma que se considere la gestión y el lanzamiento de tareas en el nuevo hardware incluido.

Por otra parte, el propio reparto del trabajo también se encuentra en manos del programador, teniendo que decidir este cómo particionar las tareas entre los dispositivos individuales y encargándose de la recolección de los resultados. Adicionalmente, como ya se argumentó en el capítulo 2, no resulta posible compartir memoria entre dispositivos pertenecientes a plataformas diferentes. Por esto, la gestión de la memoria también se convierte en una responsabilidad más en manos del programador.

Teniendo en cuenta todo lo arriba expuesto, la gestión de un entorno de trabajo heterogéneo y el lanzamiento de las tareas en OpenCL suponen una carga de trabajo considerable para el programador (creciente además con el número de dispositivos presentes en el sistema), que además es fuente potencial de errores. Por ejemplo, sólo la inicialización del hardware y la realización de las tareas necesarias para el lanzamiento del *benchmark* LavaMD, realizadas de forma manual, requieren más de 500 líneas de código si se utiliza sólo una GPU o más de 1100 si se desea aprovechar una CPU y 2 GPUs.

El objetivo de este capítulo es proponer Maat, una librería para el balanceo de carga cuyo propósito es resolver todos los inconvenientes arriba indicados. Con esto, Maat ofrece una visión del sistema completo como una unidad de cómputo a la que pueden enviarse tareas, las cuales se ejecutarán aprovechando todo el hardware disponible de forma transparente para el programador, independientemente del sistema subyacente y sin precisar ningún cambio en el código. Este reparto de tareas se realizará de forma proporcional a la capacidad de cómputo de cada dispositivo del sistema. Por otro lado, el objetivo es también facilitar al programador la gestión de un sistema heterogéneo, generando una menor cantidad de código, lo cual resulta en programas más fáciles de depurar y mantener.

4.2. Estructuras de datos

Uno de los objetivos de la librería es ofrecer al usuario la capacidad de gestionar, de forma transparente, un sistema de computación heterogéneo independientemente de los recursos que este contenga. Como es obvio una vez estudiado el estándar OpenCL, la gestión de entornos heterogéneos implica la utilización de multitud de estructuras de datos debido, entre otros motivos, a la imposibilidad de compartir estructuras entre plataformas. Así, por ejemplo, es necesario gestionar tantos contextos como plataformas o tantas colas de comandos como dispositivos. Además, salta a la vista que la cantidad de estructuras de datos a gestionar variará con el sistema en el que se ejecute el código. Teniendo en cuenta la disparidad en la cantidad de estructuras a gestionar (y el elevado número de estas) y teniendo la transparencia como objetivo, la necesidad de ocultar esta complejidad al usuario resulta inmediata. Es decir, el programador, que va a operar con el sistema como una unidad, debe trabajar también con estructuras de datos únicas. En esta sección se introducirán estas estructuras.

4.2.1. El super_contexto

En primer lugar, en OpenCL estándar, la estructura básica para la gestión de los dispositivos es el **contexto**, que contiene a los dispositivos que gestiona. El contexto es además el bloque fundamental que se utiliza en la creación del resto de estructuras a utilizar, como los *buffers* o las colas de comandos e, indirectamente, el contexto también es necesario en el lanzamiento de los *kernels*. Teniendo esto en cuenta, una vez que se considera el sistema como un todo, surge la necesidad de utilizar una estructura de datos que aglutine todos los contextos que forman un sistema (y, por consiguiente, a todos los dispositivos que lo conforman) y que pueda utilizarse como forma única de gestionar el sistema en su totalidad. Para ello, se ha definido el **super_contexto**, que es la estructura de datos a utilizar en la librería para la gestión de dispositivos disponibles en el sistema, aunque estos pertenezcan a plataformas diferentes. Esta estructura de datos es fundamental para agrupar la gestión de todos los dispositivos en un sola función y ofrecer una visión única del sistema al usuario, de acuerdo a los objetivos propuestos.

4.2.2. El super_buffer

Otra de las estructuras de datos fundamentales en OpenCL es el *buffer* que, recordando la definición presentada en el capítulo 2, es un puntero a memoria que es válido en todos los dispositivos pertenecientes a un contexto. Por consiguiente, en un entorno de trabajo heterogéneo, será preciso utilizar al menos una de estas estructuras por contexto existente[1]. Por tanto, para cumplir con el objetivo de transparencia propuesto y en analogía con la definición del *buffer* del estándar, se define el **super_buffer** como un puntero a memoria que es válido en todos los dispositivos de todos los contextos que estén contenidos en un super_contexto. Esto permite que, también en lo que se refiere a estructuras de memoria, el sistema se comporte como una unidad.

4.2.3. El super_kernel

Finalmente, en el estándar se define el kernel como la representación de una tarea a ejecutar en un dispositivo. Por consiguiente, la estructura de datos que representa a un kernel es sólo válida, al igual que los *buffers*, dentro de un contexto [2]. Por tanto y una vez más con la transparencia como meta, se define la noción de **super_kernel** como

[1] De forma efectiva, por limitaciones del driver de algunos fabricantes, es necesario utilizar, además, una de estas estructuras por GPU presente en el sistema. La compartición de *buffers* entre GPUs puede generar problemas de serialización y rendimiento degradado.

[2] Una vez más, la compartición de *kernels* entre GPUs puede generar, por limitaciones del driver de algunos fabricantes, problemas de serialización y rendimiento degradado, luego de forma efectiva es necesario un kernel por GPU.

la estructura de datos que representa a una tarea a ejecutar por los dispositivos y que tiene validez para todos los dispositivos de todos los contextos de un super_contexto. Esta definición del super_kernel posibilita que el sistema se comporte como una unidad en el lanzamiento de las tareas.

4.3. Arquitectura a nivel usuario

En general, siempre que se propone un nuevo modelo de programación se intenta que, en lo que a sintaxis y comportamiento general se refiere, no difiera mucho de las alternativas existentes y conocidas por el programador. De esta manera, se consigue minimizar el impacto de la adopción de la nueva propuesta, de modo que la curva de aprendizaje sea lo más rápida posible.

Siguiendo este criterio, para facilitar el trabajo y la adaptación a la librería, Maat se basa en OpenCL, tanto en el formato de sus funciones como en el modo general de trabajo. Esta es la opción más lógica pues, en general, un usuario interesado en utilizar T para el balanceo de carga será también es un usuario con conocimientos de OpenCL. Así, a excepción de las diferencias necesarias al ofrecer una visión del sistema como una unidad, toda la librería ha sido construida teniendo en cuenta el modo de trabajo de OpenCL estándar (Ver capítulo 2) y con la facilidad de uso como objetivo. Por este motivo, la gran mayoría de las funciones de la librería tienen su equivalente en el estándar y reciben argumentos muy similares a los de este. A continuación, tomando como referencia la estructura general de un programa OpenCL introducida en el capítulo 2, se hará un recorrido a la arquitectura de la librería.

4.3.1. Gestión del entorno de trabajo

Por definición, los primeros pasos de cualquier programa en OpenCL siempre están relacionados con el establecimiento del entorno de trabajo con el que se desea trabajar. Esto es, la detección e inicialización de los dispositivos que desean utilizarse y la creación del contexto o contextos necesarios. Así, OpenCL ofrece funcionalidades para la detección/inicialización de dispositivos individualmente. Sin embargo, una vez que se considera el sistema como una unidad de trabajo, la diferenciación entre dispositivos, y la precisión de qué dispositivo pertenece a qué plataforma resulta innecesaria. De hecho, esta diferenciación, de cara al usuario, no resulta deseable. Por este motivo, Maat permite detectar e inicializar todos los dispositivos del sistema (o de las plataformas que se deseen) y aglutinarlos en un super_contexto utilizando una única función y de forma transparente para el programador. Esta inicialización independiente de los dispositivos

subyacentes, garantiza, además, la portabilidad del código a sistemas con un *hardware* diferente, sin necesidad de realizar modificaciones. Adicionalmente, la librería también ofrece la capacidad de escoger si se desea que se utilicen las CPUs o no, así como si se desea que se utilicen únicamente los núcleos reales de estas o que se aproveche el *Hyperthreading*. Internamente y para obtener el mejor rendimiento posible en la distribución de trabajo, esta función también realiza la subdivisión de las CPUs en los núcleos que las conforman, tratándolos como dispositivos individuales.

Por otro lado, OpenCL establece la cola de comandos como forma de comunicación con los dispositivos individuales. Sin embargo, al considerar el sistema como una unidad, ya no es necesario que el usuario utilice y gestione las colas de comandos, pues no va a comunicarse en ningún momento con un dispositivo individual, sino con el sistema completo. Teniendo esto en cuenta, la misma función que crea el super_contexto también es la encargada de crear las colas de comandos necesarias para el funcionamiento adecuado del sistema. La gestión de estas colas de comandos se realiza de forma interna, sin que el usuario tenga que preocuparse por ellas. De hecho, el usuario ni tan siquiera tendrá visibilidad sobre las colas de comandos, pues utilizará siempre el super_contexto para comunicarse con el sistema.

4.3.2. Gestión de memoria

Por su parte, el *buffer* del estándar OpenCL es una estructura de datos vinculada a la plataforma, que no puede ser compartida entre dispositivos de plataformas distintas. Por tanto, para aprovechar el potencial heterogéneo del sistema necesitaremos, al menos, una de estas estructuras por plataforma existente. Por este motivo, para evitar la gestión individual de cada uno de los *buffers*, Maat ofrece, una función que permite crear un super_buffer, la cual se encarga de crear los *buffers* internos necesarios para el adecuado funcionamiento del super_buffer. Con esto. la obtención de estructuras de memoria válidas en todo el sistema resulta posible mediante el uso de una única función. Adicionalmente, la librería también dispone de las funciones necesarias para realizar lecturas y escrituras de los super_buffers de forma que la gestión de los datos en el entorno de memoria distribuida se realice de forma transparente al usuario.

4.3.3. Gestión de *kernels*

La misma filosofía se sigue para los *kernels*, que también son estructuras vinculadas a la plataforma. Así, Maat dispone de una función que permite crear super_kernels, la cual se encarga, internamente, de crear los *kernels* OpenCL que resulten necesarios para asegurar el correcto funcionamiento del super_kernel. Así, la creación de un super_kernel,

como tarea a ejecutar por los dispositivos que tiene validez en todos los contextos, se consigue mediante una llamada una única función. La librería también dispone de las funciones necesarias para asociar parámetros a los super_kernels de manera transparente y de los mecanismos necesarios para esperar a la terminación de estos.

Asimismo, Maat ofrece las funciones precisas para realizar el lanzamiento de los super_kernels. La ejecución de un super_kernel, internamente, implicará el lanzamiento (una o más veces) de un kernel estándar en cada uno de los dispositivos disponibles. Como ya se argumentó en la sección 2, los elementos fundamentales en el lanzamiento de un kernel en OpenCL estándar son el *global_work_size* y el *local_work_size*. Esto también es cierto cuando se utiliza la librería. Así, independientemente de que la ejecución del super_kernel implique la ejecución de varios *kernels* internamente en varios dispositivos, el super_kernel sólo será lanzado una única vez, utilizando el *global_work_size* y el *local_work_size* que se utilizarían con un kernel de OpenCL estándar. Es la propia librería la que, internamente, se encarga de asignarle a cada kernel individual el *global_work_size* y *global_work_offset* correspondientes para realizar un balanceo de carga correcto, respetando la lógica del programa. De esta manera, conseguimos mantener, también en el lanzamiento de los super_kernels, el objetivo de transparencia fijado. La librería ofrece funciones para realizar el lanzamiento de tareas utilizando cuatro modos de planificación distintos: Estática, dinámica, guiada y guiada heterogénea. Por su importancia en la librería, pues en estas funciones es donde se realiza el balanceo de carga al fin y al cabo, estas funciones se analizan en un apartado independiente.

4.3.4. Liberación de recursos

Finalmente, la librería también dispone de las funciones necesarias para realizar la liberación de los recursos reservados. Como es lógico de acuerdo a su comportamiento, esta liberación también se realiza de forma transparente. Esto es, la liberación de una de las estructuras de la librería supone la liberación ordenada de todas las estructuras OpenCL que esta internamente gestiona.

4.4. Estructura general de un programa utilizando Maat

Teniendo todo lo anterior en cuenta, la forma de trabajo utilizando la librería puede estructurarse en los siguientes pasos:

- Detección de las plataformas del sistema.

- Creación de un super_contexto, que contiene los dispositivos de las plataformas que se seleccionen.
- Creación de un super_kernel, el cual, por definición, es válido en todos los dispositivos y opera de forma que se eviten los problemas de rendimiento degradado.
- Creación de los super_buffers necesarios para que el super_kernel pueda operar y copia de datos. El super_buffer, por definición, es válido en todos los dispositivos y opera de forma que se eviten los problemas de rendimiento degradado.
- Asociación de parámetros al super_kernel
- Lanzamiento del super_kernel.
- Lectura de resultados una vez se ha completado la ejecución del super_kernel.
- Liberación de los super_bufferes, el super_kernel, y el super_contexto.

Como puede observarse comparando la estructura general de un programa que utilice la librería y la de OpenCL estándar (capítulo 2), efectivamente, la utilización de la librería resulta en una disminución del trabajo de programación necesario para trabajar con un sistema heterogéneo. De hecho, si se compara el número de líneas de código necesarias para lanzar una tarea en un sistema heterogéneo utilizando OpenCL estándar o Maat la diferencia es incluso más notoria. La tabla 4.1 contiene una comparativa de la cantidad de líneas de código aproximadas necesarias para lanzar el *benchmark* lavaMD en diferentes entornos. Por su parte, la tabla 4.2 contiene las equivalencias aproximadas de las funciones de la librería y las de OpenCL.

Entorno	Número de líneas
1 GPU con OpenCL	500
2 GPUs con OpenCL	800
2 GPUs y una CPU dividida en sus cores con OpenCL	1100
Sistema heterogéneo con Maat (Independientemente de los dispositivos que lo compongan y que deseen utilizarse)	500

Tabla 4.1: Comparativa de longitud de código

Maat	OpenCL
clCreateSuperContext	clCreateDevice clCreateSubDevices clCreateCommandQueue clCreateContext
clCreateSuperBuffer	clCreateBuffer
clWriteSuperBuffer	clEnqueueWriteBuffer
clReadSuperBuffer	clEnqueueReadBuffer
clCreateSuperKernel	clCreateKernel
clSetSuperKernelArg clSetSuperKernelArgSuperBuffer	clSetKernelArg
clEnqueueKernelStaticBalancing clEnqueueKernelDynamicBalancing clEnqueueKernelGuidedBalancing clEnqueueKernelHeterogeneousGuidedBalancing	clEnqueueNDRange
clReleaseSuperContext	clReleaseDevice clReleaseCommandQueue clReleaseContext
clReleaseSuperBuffer	clReleaseBuffer
clReleaseSuperKernel	clReleaseKernel

Tabla 4.2: Relación entre funciones Maat y funciones OpenCL

4.5. Lanzamiento de super_kernels y balanceo de carga

Tanto el lanzamiento de los super_kernels como el balanceo de carga son partes centrales de la librería. El equilibrio de carga implementado se basa en los datos. Esto es, no se reparten tareas diferentes a los dispositivos, sino que todos ellos realizan la misma tarea sobre datos diferentes, estando la distribución de trabajo gobernada por índices. Por tanto, al estar trabajando con un sistema de memoria distribuida, es necesario copiar de los datos de entrada a la memoria de cada dispositivo según los necesite. Por su parte, como ya se adelantó, el lanzamiento de un super_kernel se realiza de forma análoga a como se realizaría si se tratara de un kernel convencional del estándar. Esto es, sin que el hecho de que la ejecución del super_kernel, que implica el lanzamiento de una cantidad indeterminada de *kernels* individuales en diversos dispositivos, afecte a la forma en la que el usuario debe gestionar el *global_work_size* y el *local_work_size*. La gestión de estos valores para cada uno de los dispositivos que, de forma efectiva, gobierna el reparto de trabajo y el balanceo de carga, se realiza internamente. Por su parte, cada ejecución

de estos *kernels* individuales, representará un punto de sincronización con el *host* del sistema. En esta sección se explica cada uno de los métodos de planificación de trabajo y balanceo de carga que ofrece la librería, tanto desde el punto de vista del usuario que va a utilizarlos y que sólo debe preocuparse de su función y forma de uso, como de su implementación interna. La figura 4.1, al final de esta sección, ilustra el comportamiento de cada uno de estas técnicas de equilibrio de carga.

4.5.1. Balanceo estático

La opción de balanceo estático realiza un reparto de la carga de trabajo completa en el momento de lanzar la ejecución. Es decir, el lanzamiento de un super_kernel con balanceo estático supone, por cada dispositivo, la asignación de una porción de la carga de trabajo y el lanzamiento de un kernel OpenCL. Por consiguiente, de forma efectiva, se lanzarán tantos *kernels* OpenCL como dispositivos (uno por dispositivo), con una carga de trabajo agregada igual a la carga total de trabajo. Este reparto de la carga se realiza tomando como base la potencia de cómputo de cada uno de los dispositivos presentes en el sistema, que es un dato que debe ser estático del sistema. Así, se asigna a cada dispositivo una cantidad de trabajo proporcional a su potencia en relación con la potencia total del sistema. La fórmula utilizada para realizar el reparto es la siguiente:

$$w_i = \frac{P_i}{P_T} \cdot w_T$$

Donde w_T y w_i son el trabajo total y por dispositivo respectivamente y P_T y P_i la potencia total del sistema y por dispositivo. Esto, internamente, se gestiona dividiendo el *global_work_size* en partes proporcionales a la potencia de cada dispositivo y asignándole a cada dispositivo el *global_work_size* y el *global_work_offset* que le corresponda, de modo que se respete la lógica del algoritmo si este se hubiera ejecutado en un único lanzamiento de kernel y en un único dispositivo. Con todo esto, la ejecución del super_kernel habrá terminado una vez haya terminado la ejecución de cada uno de los *kernels* individuales.

En esta técnica de balanceo, la carga de trabajo es estática, esto es, no se modifica a lo largo de la ejecución. Además, sólo requiere un punto de sincronización entre los dispositivos y el *host* (al terminar cada uno de ellos su única ejecución de*kernel*). Por estos motivos, esta es una opción sencilla y eficiente en cargas homogéneas, pero presenta problemas cuando la carga es heterogénea, al utilizar una carga de trabajo fija. Además, presenta el inconveniente de requerir que se determine la potencia de cómputo de los

dispositivos, parámetro no sólo dependiente de la aplicación, sino del sistema en el que esta esté ejecutándose.

4.5.2. Balanceo dinámico

En contraposición con el caso estático, en el lanzamiento dinámico de un super_kernel no se realiza un reparto de la carga de trabajo completa en el momento de lanzar la ejecución. Por contra, el trabajo se divide en porciones muy pequeñas y de idéntico tamaño (habrá muchas más porciones de trabajo que dispositivos), las cuales irán ejecutándose en los dispositivos que se encuentren libres en cada momento. Con esto, al lanzar un super_kernel de forma dinámica, internamente, también se producirá el lanzamiento de un kernel por dispositivo como en el caso estático, pero la carga agregada de estos *kernels* lanzados en primera instancia será inferior a la carga total. Así, tras este lanzamiento de *kernels* inicial y mientras queden porciones de trabajo por ejecutar, una vez un dispositivo termine una ejecución, recibirá automáticamente otra porción de trabajo, la cual resultará en una nueva ejecución del kernel OpenCL. Con todo esto, el lanzamiento dinámico de un super_kernel resultará, internamente, en el lanzamientos de tantos *kernels* OpenCL como porciones en las que se haya dividido el trabajo. Además, no resulta posible predecir cuánto trabajo va a ser ejecutado por cada dispositivo, pues las porciones de trabajo son asignadas a los dispositivos según estos se quedan libres.

El número de partes en las que debe dividirse el trabajo es una cantidad que debe ser fijada por el usuario o que se escoge de manera automática, encargándose las funciones de la librería de realizar todas las tareas relativas al balanceo dinámico de forma autónoma. Al igual que en el caso estático, la asignación de una porción de trabajo a un dispositivo está gobernada por el *global_work_size* y el *global_work_offset* de cada lanzamiento de kernel individual. Teniendo en cuenta, como se introdujo en el capítulo 2, que el *global_work_size* debe ser múltiplo del *local_work_size*, lo cual implica que la mínima unidad de trabajo ejecutable por un dispositivo es un *local_work_size*, el número de partes en las que se debe dividir el trabajo no puede exceder *global_work_size*/*local_work_size*

Esta técnica, al contrario que la estática, se comporta bien frente a cargas de trabajo heterogéneas y entornos dinámicos. Sin embargo, presenta el inconveniente de implicar multitud de puntos de sincronización innecesarios, pues los dispositivos se sincronizarán con el *host* cada vez que completen una ejecución de *kernel*.

4.5.3. Balanceo guiado (*guided*)

Al igual que en el balanceo dinámico, en el lanzamiento guiado de un super_kernel tampoco se realiza un reparto de la carga de trabajo completa en el momento de lanzar la ejecución. El balanceo guiado podría considerarse una balanceo dinámico en el que la cantidad de trabajo que se asigna a cada dispositivo disminuye según avanza la ejecución. De esta manera, los paquetes de trabajo se hacen más pequeños al llegar el final de la ejecución. Así, de forma idéntica a como se comporta la opción dinámica, el lanzamiento de un super_kernel de forma guiada resulta en el lanzamiento de un kernel por dispositivo, con una carga agregada inferior a la carga de trabajo total, seguida de más ejecuciones de kernels según los dispositivos se queden libres. La diferencia, como se ha adelantado, es que cada una de estas ejecuciones (también las iniciales) recibe una carga de trabajo diferente y decreciente.

En concreto, la carga de trabajo asignada es proporcional a la cantidad de trabajo restante, dividida entre el número de dispositivos presentes en el sistema. Así, la cantidad de trabajo asignada a cada dispositivo disminuye sucesivamente hasta un valor mínimo, el cual debe ser fijado por el programador en el lanzamiento del kernel. Como se argumentó en la subsección anterior, la mínima unidad de trabajo ejecutable por un dispositivo es un *local_work_size*, con lo que el valor mínimo hasta el que debe disminuir la cantidad de trabajo a asignar a los dispositivos, se especifica como un entero que representa el número de estas unidades mínimas a ejecutar en la asignación de trabajo más pequeña. Al igual que en las otras formas de balanceo, la asignación de una porción de trabajo a un dispositivo está gobernada por el *global_work_size* y el *global_work_offset* de cada lanzamiento de kernel individual.

La principal ventaja de esta técnica de balanceo es que se disminuye la cantidad de puntos de sincronización entre dispositivos y *host*, manteniendo la adaptabilidad de un método de balanceo dinámico. Además, el reparto del trabajo con grano fino se realiza al final, que es cuando resulta necesario. Sin embargo, puede surgir el problema de que, si existe una gran diferencia de rendimiento entre los dispositivos, se asigne inicialmente una porción de trabajo demasiado grande al dispositivo lento, de forma que el rápido consuma todo el trabajo restante antes de que el lento termine y la aplicación tenga que esperarle. Esta situación, por norma general, se produce entre la GPU y la CPU, pues la primera suele tener una potencia de cómputo mucho mayor que la segunda.

4.5.4. Balanceo guiado heterogéneo

La cuarta opción en el lanzamiento de *kernels*, el balanceo guiado heterogéneo, actúa de manera similar al balanceo guiado, pero la cantidad de trabajo asignada a cada

dispositivo depende además de si este es una CPU o una GPU. Por tanto, una vez un dispositivo se quede libre, este recibirá una porción de trabajo proporcional a dos factores: la cantidad de trabajo restante, dividida entre el número de dispositivos del sistema (planificación *guided* básica) y a un factor de corrección. Este factor de corrección se calcula, una vez más, en base al concepto de potencia de dispositivo introducido en la planificación estática.

$$w_i = \frac{P_i}{P_T} \cdot \frac{w_r}{n}$$

Donde w_r y w_i representan el trabajo restante y por dispositivo respectivamente, P_T y P_i la potencia total del sistema y por dispositivo y n el número de dispositivos presentes en el sistema. Con todo esto, el objetivo de este método de balanceo es asignar una menor cantidad de trabajo a aquellos dispositivos que tengan menor potencia, de modo que todos estén la mayor parte del tiempo ocupados, al mismo tiempo que se minimice la cantidad de puntos de sincronización.

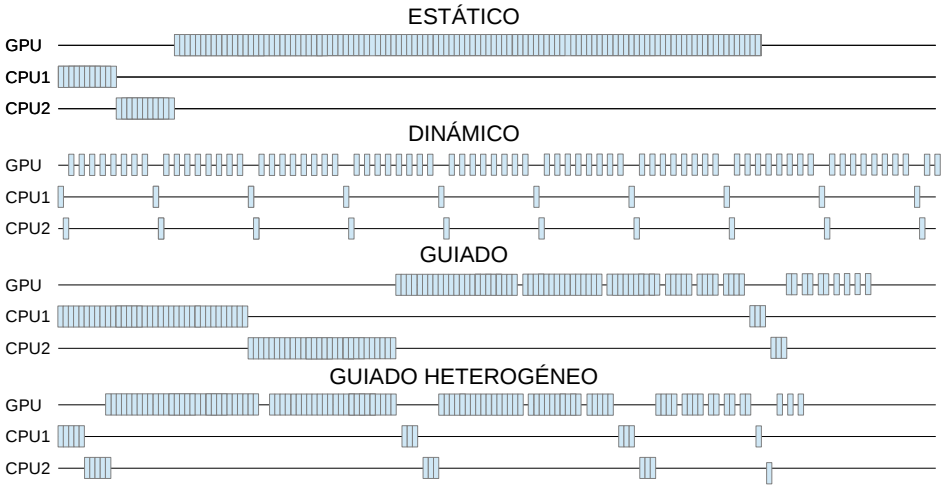

FIGURA 4.1: Comportamiento de las técnicas de balanceo

Capítulo 5

Validación experimental

Habiendo cumplido el objetivo de facilitar la labor de programación, ofreciendo una visión unitaria del sistema, la librería aún tiene que demostrar su solvencia respecto a otro factor: Que su utilización suponga una mejora en el rendimiento de las aplicaciones. Para demostrar esto, es preciso realizar una serie de pruebas, utilizando los escenarios adecuados y tratando de capturar las métricas de rendimiento que resulten de interés para el problema. El propósito de este capítulo es explicar la metodología seguida a la hora de realizar estas pruebas, así como introducir el sistema utilizado para la experimentación y presentar los resultados obtenidos.

5.1. Entorno de experimentación

Las pruebas de rendimiento han sido realizadas en un sistema heterogéneo que dispone de dos CPUs, dos GPUs y 16GBs de memoria RAM. Cada CPU es una Intel Xeon E5-2620, la cual consta de 6 núcleos a 2GHz, capaces de ejecutar 2 *threads* cada uno mediante *Hyperthreading*, por lo que, en total, se dispone de 12 núcleos físicos y 24 lógicos. Ambas CPUs están conectadas vía QPI, lo cual hace que OpenCL detecte un único dispositivo que comprende a las dos. Por este motivo, en lo que resta de capítulo, nos referiremos a la CPU como el dispositivo que engloba a los dos procesadores disponibles. Por su parte, las GPUs son dos Tesla K20m de Nvidia, cada una con 13 vías de ejecución SIMD, que son capaces de ejecutar 2048 *threads* cada una. Adicionalmente, cada una de las GPUs posee un *slot* PCI Express versión 2.0 independiente. El sistema utiliza el sistema operativo CentOS en su versión 6 y el *driver* de OpenCL de Intel.

Por otra parte, si se desea obtener unos resultados de calidad, que resulten útiles para verificar el cumplimiento de los objetivos propuestos, es necesario establecer unas bases

antes de iniciar la experimentación. Estos aspectos fundamentales a fijar previamente son:

1. Escoger las aplicaciones a utilizar
2. Fijar los parámetros de rendimiento a analizar
3. Establecer la metodología de experimentación
4. Obtener unos tiempos de referencia (*baseline*)
5. Establecer los parámetros propios de cada técnica de balanceo

La selección de las aplicaciones más apropiadas para el escenario de este proyecto es tratada en el capítulo 3 de esta memoria. De acuerdo a lo argumentado en ese capítulo, para realizar las pruebas experimentales se ha utilizado el *benchmark* LavaMD y una implementación del algoritmo de resolución del problema RAP. Asimismo, las métricas de rendimiento a analizar, por ser representativas a la hora de escoger las aplicaciones para la experimentación, también son discutidas en el capítulo 3. Estas son:

- **Tiempo total de ejecución**, incluyendo inicialización, ejecución del kernel, liberación de recursos y comunicación. Este último aspecto, como se argumentó, es limitante en el rendimiento de las aplicaciones (sobre todo en el caso de las GPUs) y muchas veces es ignorado en los artículos.
- **Tiempo de kernel**, que incluye el tiempo invertido en ejecución paralela en los dispositivos, así como el tiempo necesario para realizar el reparto de la carga entre estos.
- **Tiempo de comunicación**. Por su relevancia, este tiempo merece un análisis independiente. Se ha considerado sólo el tiempo asociado a la escritura de los parámetros de entrada en los dispositivos, pues el tiempo de lectura de resultados es insignificante.

Para obtener las métricas de rendimiento, al igual que a la hora de realizar la caracterización de los *benchmarks*, se han realizado, tres ejecuciones con diversos tamaños de problema, calculando la media de cada métrica para cada tamaño. De esta forma, se disminuye el impacto de posibles resultados anormales sobre los parámetros utilizados para valorar el rendimiento. En cualquier caso, se ha observado que los tiempos de ejecución de las aplicaciones, para un tamaño dado, tienen una desviación insignificante.

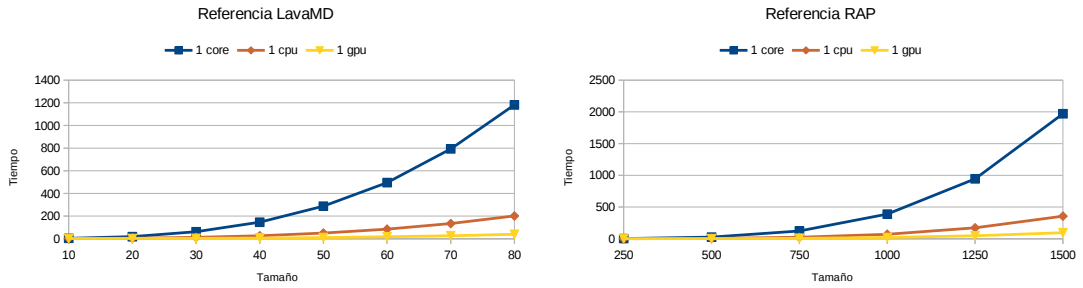

FIGURA 5.1: *Baseline*

Por otro lado, es necesario establecer unos tiempos de referencia con los que comparar los resultados obtenidos utilizando la librería (*baseline*). De esta manera, se utilizará este *baseline* para valorar la mejora de rendimiento que resulta de utilizar cada una de las técnicas de balanceo de carga propuestas. Para conseguir esta referencia, se han realizado ejecuciones, de acuerdo con la metodología anterior, de las aplicaciones sobre una GPU, una CPU (formada por 12 cores) y un core, utilizando OpenCL estándar.

De acuerdo a los resultados contenidos en la figura 5.1, el mejor rendimiento se obtiene utilizando la GPU, siendo el tiempo total de ejecución de esta del orden de 6 veces superior que el de la CPU en lavaMD y 4 veces superior en RAP. Esta diferencia de rendimiento se produce cuando el problema tiene un tamaño considerable, pues con tamaños pequeños el rendimiento es similar en todos los dispositivos. Como era esperable, existe una gran diferencia entre el rendimiento utilizando un único core o la CPU al completo. Sin embargo, cabe destacar que el tiempo de un core, tanto en lavaMD como en RAP es sólo del orden de seis veces superior al de la CPU completa cuando esta dispone de doce cores.

Como parte del *baseline*, también resulta interesante calcular el *speedup* teórico que puede obtenerse de cada aplicación teniendo en cuenta el *hardware* a utilizar. Para ello, se calcula la potencia relativa de los dispositivos lentos (cores) con respecto de los rápidos (GPUs) y se obtiene la potencia total del sistema normalizada al dispositivo más rápido. De esta manera, esta potencia total normalizada coincide con el *speedup* máximo teórico sobre el tiempo del dispositivo más rápido, que es la GPU. Los valores para cada aplicación pueden consultarse en la tabla 5.1.

	LavaMD	RAP
Potencia de GPU	1	1
Potencia de core normalizada	0,0329	0,048
speedup teórico	2,3952	2,5763

TABLA 5.1: Potencias normalizadas y *speedup* teórico por aplicación

De acuerdo a los datos de la tabla, existe una diferencia de rendimiento enorme entre el rendimiento de una GPU y el de un core. En definitiva, el sistema completo es equivalente a utilizar 2,39 GPUs para LavaMD y 2,57 para RAP, con lo cual estas serán las cotas máximas de mejora en el caso del tiempo total.

Finalmente, es necesario definir los valores de los parámetros propios de cada una de las técnicas de balanceo de carga de la librería. En primer lugar, es necesario obtener la potencia de los dispositivos (CPU y GPUs), la cual es necesaria en el balanceo estático y guiado heterogéneo. Considerado el *baseline* obtenido, este parámetro se calcula normalizando la potencia de cómputo respecto a la del dispositivo más lento. Por otro lado, también debe definirse la cantidad de paquetes a utilizar en el balanceo dinámico, así como el tamaño mínimo de paquete para ambos métodos guiados. Estos parámetros han sido fijados experimentalmente, realizando una serie de ejecuciones previas, en primer lugar de grano grueso y después más fino, para afinar su valor hasta obtener el mejor rendimiento posible. Estos parámetros han sido establecidos tratando de maximizar el rendimiento para tamaños grandes de problema, pues es en estos tamaños en los que tiene sentido realizar equilibrio de carga. Esto se debe a que, si el problema es pequeño, la sobrecarga asociada a la distribución de la carga resultaría en un rendimiento inferior al que se obtendría de utilizar sólo el dispositivo más potente del sistema. En cualquier caso, en los resultados experimentales se presenta un barrido con diferentes tamaños de problema para poder apreciar la evolución de los resultados. La tabla 5.2 contiene los parámetros seleccionados para cada aplicación. Cabe destacar que el tamaño mínimo de paquete se refiere a cuantas unidades mínimas de trabajo componen el paquete (recordemos que, de acuerdo a lo argumentado en el capítulo 4, la unidad mínima de trabajo es un *local_work_size*)

	LavaMD	RAP
Potencia de CPU	1	1
Potencia de GPU	30	21
Número de paquetes (Sistema completo)	500	200
Número de paquetes (CPU)	500	8000
Número de paquetes (2 GPUs)	500	500
Tamaño mínimo de paquete (Sistema completo)	5	200
Tamaño mínimo de paquete (CPU)	1	200
Tamaño mínimo de paquete (2 GPUs)	20	200

Tabla 5.2: Parámetros de balanceo por aplicación

Además, con objeto de verificar las capacidades de reparto de trabajo de la librería, se han realizado pruebas con equilibrio de carga que considera sólo la CPU, repartiendo el

trabajo entre los 12 núcleos que la forman, sólo las 2 GPUs o el sistema al completo. Por claridad a la hora de representar los datos en gráficas, salvo en las de tiempo de comunicación, sólo se han realizado comparaciones con los datos de referencia que resultan significativos en cada caso. Es decir, en el caso del balanceo sobre la CPU, se ha comparado el tiempo obtenido con el *baseline* de la CPU, mientras que en el resto de los casos se ha comparado con el tiempo de referencia de la GPU. Además, en los casos en los que no se utiliza el sistema completo, al considerar sólo dispositivos homogéneos, el balanceo guiado heterogéneo no tiene sentido, pues, al tener todos los dispositivos la misma potencia, este balanceo equivale al guiado básico. Por este motivo, el balanceo guiado heterogéneo sólo se ha considerado cuando se utiliza el sistema completo. Por la misma razón, en sistemas heterogéneos no se ha considerado el balanceo guiado básico por no tener sentido de acuerdo a lo argumentado en el capítulo 4 (el hecho de, potencialmente, tener que esperar a un dispositivo lento, la CPU habitualmente, al que se le haya asignado una porción de trabajo grande al principio de la ejecución). Las siguientes secciones se centran en presentar y analizar los resultados obtenidos para cada aplicación.

5.2. Resultados lavaMD

5.2.1. Sistema MultiCPU-MultiGPU

Como puede observarse, de acuerdo a los resultados obtenidos (Figura 5.2), los métodos de equilibrio de carga generan un tiempo de ejecución total superior al tiempo de referencia para tamaños de problema pequeños. Sin embargo, para tamaños grandes, las técnicas de Maat rinden alrededor de 1,7 veces mejor que la versión de referencia que utiliza una sola GPU.

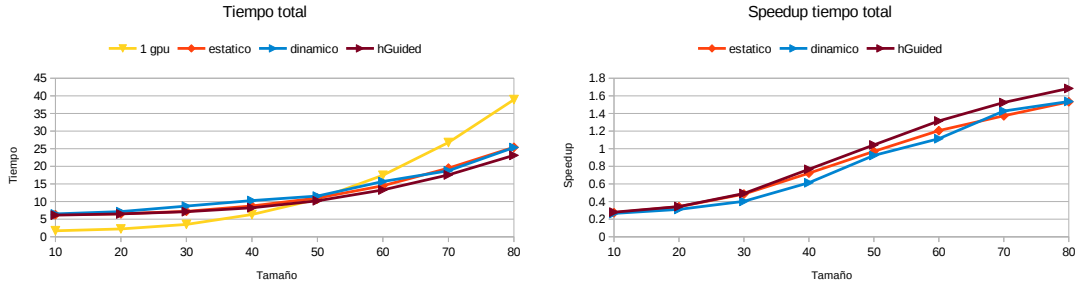

FIGURA 5.2: Comparativa del tiempo total en LavaMD

En general y debido a la naturaleza homogénea de la propia aplicación, no hay grandes diferencias entre el rendimiento utilizando cada uno de los métodos de equilibrio de carga. El comportamiento de cada método coincide con lo esperado en el análisis teórico

del capítulo 4. El balanceo dinámico presenta un rendimiento ligeramente peor que el estático, debido al exceso de puntos de sincronización, para todos los tamaños de problema salvo para el más grande. Esto puede indicar que, para tamaños de problema mayores, este método escalará mejor. Por su parte, el balanceo guiado heterogéneo, pese a la naturaleza homogénea de la aplicación, al tener en cuenta la potencia de cómputo de cada dispositivo y minimizar los puntos de sincronización, manteniendo una planificación dinámica, es el que presenta el mejor rendimiento. La tabla 5.3 contiene los *speedups* para cada método de balanceo de carga con el tamaño de problema más grande.

Método de balanceo	*Speedup*
Estático	1,532
Dinámico	1,536
Guiado heterogéneo	1,68

Tabla 5.3: Speedups LavaMD

El hecho de que la librería rinda peor que la referencia para tamaños de problema pequeños se debe a un *overhead* observado al realizar las pruebas de alrededor de 6 segundos, como puede apreciarse en la figura 5.2. Las posibles causas de este *overhead*, que se analizarán más adelante, son:

- El tiempo que es necesario invertir en el propio algoritmo de balanceo de carga
- El tiempo de comunicación
- El tiempo de inicialización del driver y de los dispositivos

La figura 5.3 presenta los resultados del tiempo de *kernel*, el *speedup* obtenido para tamaños pequeños con los métodos dinámico y guiado heterogéneo es inferior a 1. Esto se debe a que, por la poca carga del problema, la sobrecarga introducida con el balanceo de carga impide mejorar el rendimiento. En el caso del balanceo dinámico, este hecho también se debe a que se haya fijado la cantidad de paquetes teniendo como objetivo maximizar el rendimiento para tamaños de problema grandes, luego esta cantidad es excesiva en tamaños pequeños. Podría haberse obtenido un mejor rendimiento con este método si se hubiera fijado una cantidad de paquetes inferior. Por su parte, sin ser el que produce el mayor *speedup* (guiado heterogéneo), el balanceo estático se comporta bien tanto para tamaños pequeños como grandes, con una mejora de rendimiento muy constante. Cabe destacar también que, si consideramos únicamente el tiempo de *kernel*, el *speedup* al utilizar Maat aumenta, alcanzando un valor de 2,2 en el caso estático y 2,55 en el caso guiado heterogéneo. Por otro lado, también se aprecia que el método

de balanceo que mejor se adapta a los tamaños de problema pequeños es el estático. Con esto, queda descartado que la causa del mal rendimiento para tamaños pequeños se deba a un *overhead* relacionado con la distribución de la carga entre los dispositivos del sistema pues, si bien es cierto que tiene cierto impacto para tamaños pequeños de problema, no se justifica la sobrecarga de 6 segundos observada.

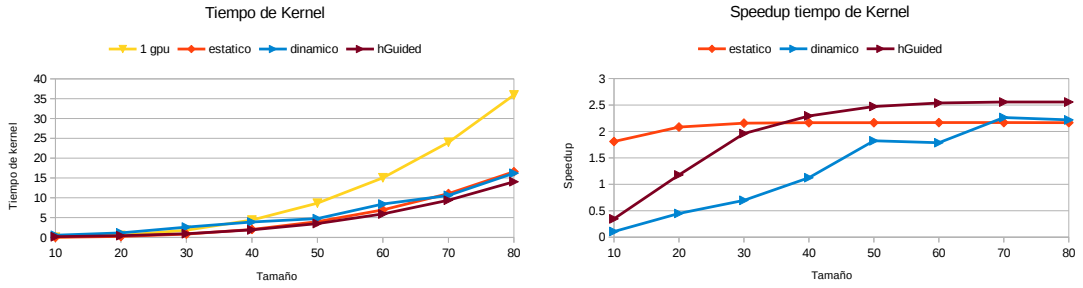

FIGURA 5.3: Comparativa del tiempo de *kernel* en LavaMD

Por su parte, la figura 5.4 compara el tiempo de comunicación en el entorno de referencia con el que se obtiene al utilizar la librería. No se diferencia el tiempo de comunicación para cada técnica de balanceo, pues la comunicación es independiente del método seleccionado. Como puede observarse, el aprovechamiento del potencial heterogéneo del sistema tiene cierto impacto sobre el tiempo de comunicación, al ser necesario copiar los parámetros de entrada del problema a una mayor cantidad de dispositivos. En general, el tiempo asociado a la comunicación con los dispositivos se triplica. Este hecho es lógico teniendo en cuenta que, por su construcción, LavaMD requiere que todos los dispositivos posean una copia completa de los datos de entrada del programa. Habría sido posible modificar la aplicación de modo que pudiera evitarse la copia de los datos a todos los dispositivos, pero esto habría ido en contra del principio introducido en el capítulo 3 de realizar las pruebas de rendimiento utilizando aplicaciones estándar. Por su parte, el impacto sobre el tiempo de comunicación es insignificante para tamaños de problema pequeños, con lo cual el *overhead* inicial tampoco está relacionado con el tiempo de comunicación.

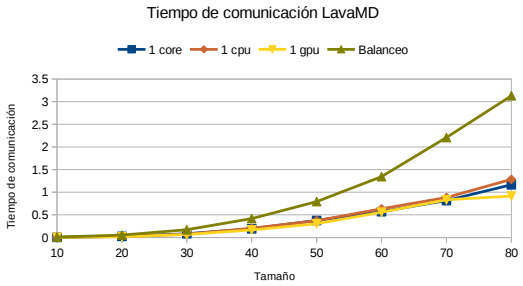

FIGURA 5.4: Comparativa del tiempo de comunicación en LavaMD

Descartados el tiempo de comunicación y de *kernel*, se procede a analizar el tiempo de inicialización del *driver*. El análisis determina que, efectivamente, este es el origen del *overhead* pues, independientemente del tamaño de problema, el arrancar el *driver* y crear el programa para cada dispositivo, requiere alrededor de 6 segundos. En concreto, este tiempo depende de la cantidad de CPUs presentes en el sistema, pues la inicialización del *driver* para cada una de ellas es de aproximadamente medio segundo (recordemos que cada core es considerado como una CPU de pleno derecho). En cualquier caso, al tomar un valor fijo e independiente del tamaño de problema, el impacto del tiempo de inicialización sobre el tiempo total disminuiría al aumentar el tamaño del problema. Sin embargo, el *benchmark* LavaMD no permite utilizar tamaños superiores a 80. Otra opción, la cual eliminaría el problema por completo, es mantener el *driver* de los dispositivos inicializado. El análisis del tiempo de inicialización y su tratamiento quedan como trabajo futuro.

En definitiva, la distribución de la carga del *benchmark* LavaMD entre todos los dispositivos de un sistema heterogéneo genera buenos resultados. Teniendo en cuenta el impacto relativamente bajo del balanceo sobre el tiempo de comunicación y que el inconveniente de la inicialización del *driver* es solucionable, podrían alcanzarse *speedups* cercanos al calculado sobre el tiempo de *kernel*, que es de 2,55. Este es un muy buen resultado si se considera que el *speedup* teórico del sistema para LavaMD (sobre el tiempo total) es de 2,39 (Tabla 5.1). Por tanto, puede afirmarse que el máximo teórico es alcanzable utilizando las técnicas de balanceo propuestas en la librería.

5.2.2. Sistema MultiCPU

Como se observó al calcular el *baseline* de LavaMD, el rendimiento utilizando la CPU completa sólo es 6 veces superior al obtenido utilizando un único núcleo. Sin embargo, la CPU dispone de 12 núcleos. Por este motivo, resulta de interés comprobar el rendimiento de la librería al distribuir el trabajo entre los 12 núcleos de la CPU. El resultado, como puede apreciarse en la figura 5.5, es que el balanceo de carga estático ofrece aproximadamente el mismo rendimiento que el reparto realizado internamente por OpenCL, mientras que el guiado sólo lo mejora ligeramente. Por su parte, el balanceo dinámico, para tamaños pequeños, al igual que en el caso anterior, presenta un *speedup* menor que uno por haberse fijado la cantidad de paquetes con los tamaños de problema grandes como objetivo. Sin embargo, según aumenta el tamaño de problema, el *speedup* sobre el tiempo de referencia obtenido con OpenCL estándar, crece de forma aproximadamente lineal, hasta alcanzar un máximo de 2. Con lo cual, el uso de la librería en modo dinámico permite alcanzar un *speedup* de alrededor de 12 al utilizar la CPU completa (12 núcleos) frente a utilizar sólo uno. En definitiva, este es un muy buen resultado pues,

teniendo en cuenta el hardware disponible, el utilizar Maat supone une mejora sustancial de la eficiencia, pues se consigue aprovechar el 100 % de la CPU frente al 50 % obtenido utilizando OpenCL.

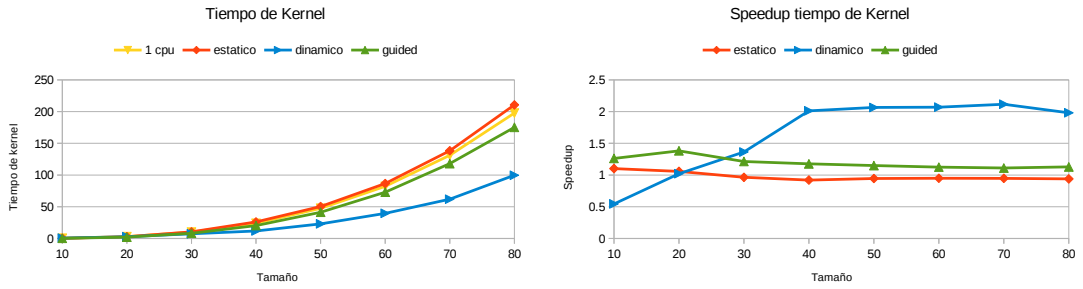

FIGURA 5.5: Comparativa del tiempo de *kernel* en LavaMD con una CPU

5.2.3. Sistema MultiGPU

La obtención de un buen rendimiento mediante el uso de balanceo de carga depende de la capacidad de la librería para conseguir que todos los dispositivos estén ocupados la mayor parte del tiempo independientemente de su potencia de cómputo. Esto es, cabe la posibilidad de que, si el reparto de trabajo no se realiza de forma adecuada, añadir dispositivos provoque una pérdida de rendimiento, al tener que esperar los dispositivos rápidos a los lentos. Por tanto, resulta de interés verificar la ganancia de rendimiento obtenida utilizando sólo las GPUs, de forma que se pueda comprobar si se ha realizado un equilibrio de carga correcto. Como puede apreciarse en la figura 5.6, el utilizar 2 GPUs supone obtener un *speedup* de 2 frente al tiempo obtenido con una sola GPU. Con lo cual, la librería es capaz de distribuir adecuadamente el trabajo entre ambas GPUs, aprovechando totalmente la potencia que ofrecen. En la figura 5.6 no se ha detallado el tiempo por modo de balanceo por haberse obtenido resultados prácticamente idénticos con todos ellos.

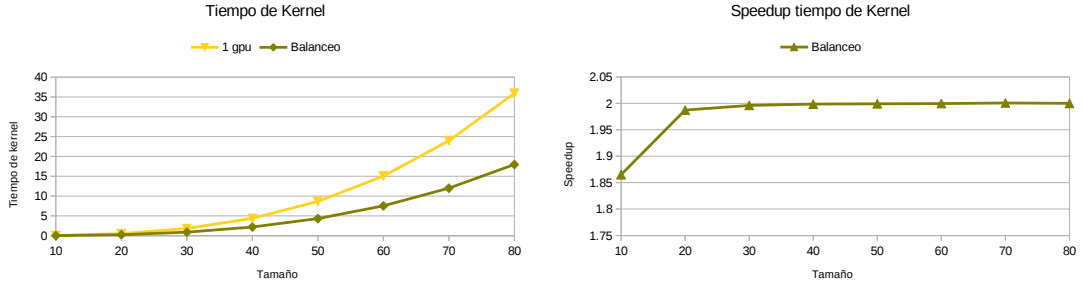

FIGURA 5.6: Comparativa del tiempo de *kernel* en LavaMD con 2 GPUs

5.3. Resultados RAP

5.3.1. Sistema MultiCPU-MultiGPU

De acuerdo a los datos presentados figura 5.7, las técnicas de equilibrio de carga de Maat mejoran en mayor o menor medida el tiempo obtenido utilizando una sola GPU para los tamaños grandes de problema. Para los tamaños pequeños, al igual que en el caso de LavaMD, la ganancia de rendimiento no compensa la sobrecarga introducida por el balanceo de carga. Es destacable que, en este caso, no aparece el *overhead* asociado a la inicialización del *driver* y la creación del programa para los dispositivos. Esto se debe a que el *kernel* asociado a RAP es mucho más simple y corto que el de LavaMD, luego la inicialización y creación del programa son también más breves. Cabe recordar que esta aplicación se caracteriza por su heterogeneidad. Esto es, cada paquete de trabajo representa una carga diferente (creciente según avanza la ejecución del programa).

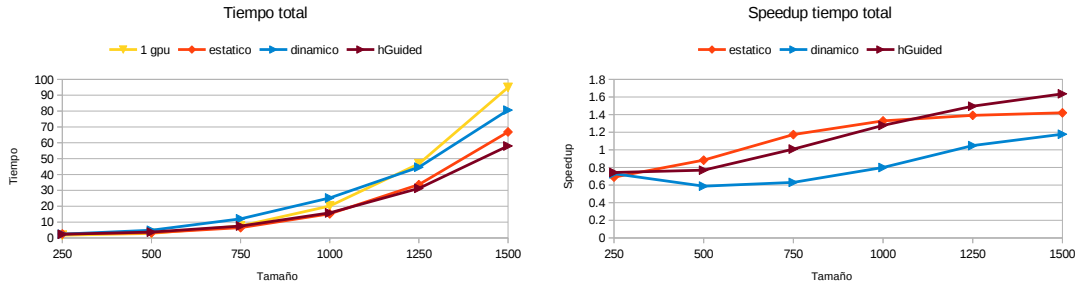

FIGURA 5.7: Comparativa del tiempo total en RAP

Con el balanceo dinámico se obtiene un *speedup* bajo. Esto se debe al carácter heterogéneo de la aplicación, que resulta en que, pese a la distribución dinámica de la carga, se presenten desbalanceos que hacen que unos dispositivos terminen antes que otros, degradando el rendimiento. Además, como los últimos paquetes tienen una carga de trabajo mucho mayor que los primeros, la asignación de uno de los últimos a un core retrasará la ejecución de toda la aplicación. De nuevo, podrían obtenerse mejores resultados con este método para los tamaños pequeños de problema utilizando una cantidad de paquetes diferente, pues la utilizada tiene como objetivo maximizar el rendimiento en tamaños grandes. En general, la causa del pobre rendimiento del balanceo dinámico es que, para evitar que las CPUs reciban paquetes demasiado grandes que retrasen la terminación de la ejecución, es necesario dividir la carga en muchos paquetes muy pequeños, pero esto también implica gran cantidad de puntos de sincronización, lo cual hace que el rendimiento se resienta. Precisamente el carácter heterogéneo de la aplicación podría hacer pensar que el balanceo estático produciría malos resultados. Sin embargo, los resultados de este método de balanceo son mejores que los del equilibrio de carga dinámico. La causa de esto es que, tal y como está implementada la librería, se asigna

trabajo a las CPUs antes que a las GPUs y, teniendo en cuenta que en RAP la carga de trabajo crece según avanza el problema, el resultado es que las porciones más largas son asignadas a las GPUs. Sin embargo, este resultado podría variar sustancialmente si la aplicación tuviera otro comportamiento. Al igual que en LavaMD, al distribuir la carga dinámicamente teniendo en cuenta la potencia de cómputo de los dispositivos y minimizando los puntos de sincronización, el balanceo guiado heterogéneo es el que mejor rendimiento produce. La tabla 5.4 contiene el *speedup* obtenido para cada técnica de balanceo de carga con el tamaño de problema más grande.

Método de balanceo	Speedup
Estático	1,42
Dinámico	1,17
Guiado heterogéneo	1,636

Tabla 5.4: Speedups RAP

En el caso de esta aplicación, al no existir el *overhead* relacionado con la inicialización, la gráfica relativa al tiempo de *kernel* es muy similar a la del tiempo total. La única diferencia es que, al no tomar en cuenta los tiempos de comunicación, si calculásemos los *speedups* sobre tiempo de *kernel* estos serían ligeramente mayores. Al no aportar ninguna información adicional, esta gráfica se ha omitido.

Por último, la figura 5.8 realiza una comparativa entre los tiempos de comunicación de referencia y los obtenidos utilizando las técnicas de balanceo. Una vez más, el tiempo de comunicación no es dependiente del modo de balanceo usado. Al igual que en el caso anterior, puede apreciarse que repartir la carga entre todos los dispositivos del sistema supone triplicar el tiempo de comunicación. Esto se debe, una vez más, a necesidades del *kernel* utilizado, que requiere que todos los dispositivos dispongan de los datos de entrada al completo. Una modificación del *kernel* podría permitir reducir el tiempo de comunicación. Esta mejora queda como trabajo futuro.

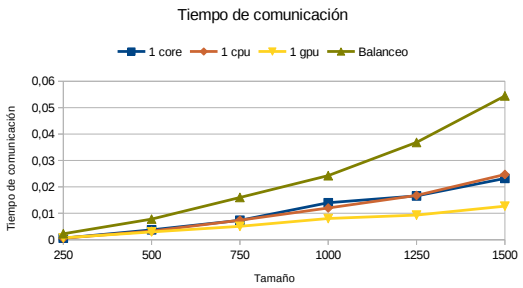

Figura 5.8: Comparativa del tiempo de comunicación en RAP

En definitiva, la utilización de la librería con la aplicación RAP resulta en una mejora de rendimiento, aunque no tan buena como en el caso de LavaMD. La causa de esto es el carácter heterogéneo de la aplicación, que dificulta aprovechar todos los recursos del sistema.

5.3.2. Sistema MultiCPU

La figura 5.9 contiene el resultado de distribuir la carga del problema RAP entre los 12 núcleos de la CPU. Como puede apreciarse, el método de balanceo estático produce un *speedup* siempre inferior a uno. La causa de esto es la heterogeneidad de la aplicación pues, de asignarle la misma cantidad de trabajo a cada núcleo, aquellos a los que les han correspondido las últimas porciones de trabajo tardarán mucho más tiempo en completar la ejecución, resultando esto en que la carga esté desequilibrada y los recursos infrautilizados. Por su parte, el método guiado, cuyo tamaño de paquete disminuye según avanza la ejecución, consigue evitar el problema que padece el balanceo estático e igualar el rendimiento de OpenCL. Por último, como en el caso de LavaMD para la CPU, el método dinámico consigue un *speedup* de 2 con respecto del tiempo obtenido utilizando OpenCL (para tamaños grandes). En el caso de tamaños pequeños, el *speedup* es inferior a uno. El motivo, una vez más, es que la cantidad de paquetes ha sido fijada de forma que se maximice el rendimiento para tamaños grandes, pero podrían obtenerse resultados similares en tamaños pequeños fijando una cantidad de paquetes menor. Cabe destacar, como demuestra la gráfica, que RAP es muy sensible a la cantidad de paquetes seleccionada. Un *speedup* de 2 sobre el tiempo obtenido en OpenCL utilizando el mismo *hardware* es un gran resultado, pues permite mejorar la eficiencia en el uso de la CPU de un 50 % a un 100 % (al considerar que la CPU tiene 12 cores y el *speedup* obtenido con OpenCL sobre 1 core es sólo de 6).

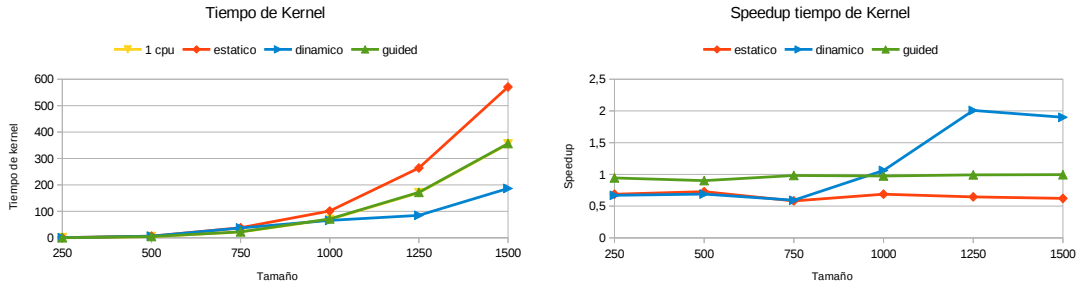

FIGURA 5.9: Comparativa del tiempo de *kernel* en RAP con una CPU

5.3.3. Sistema MultiGPU

Por último, la figura 5.10 contiene el tiempo de *kernel* para 2 GPUs utilizando las técnicas de equilibrio de carga, en comparación con el *baseline* para 1 GPU, así como el *speedup* obtenido. Considerando los resultados, puede apreciarse que los métodos dinámico y guiado no se comportan bien con tamaños pequeños (sobre todo el dinámico). Al igual que en el apartado anterior, puede apreciarse que la aplicación, por su heterogeneidad, es muy sensible a la cantidad de paquetes en las que se divide el trabajo. Por este motivo el *speedup* es mayor que 1 sólo en los tamaños más grandes. Por su parte, el método estático genera un *speedup* constante independientemente del tamaño de problema, aunque es bastante pequeño (alrededor de 1,3). Este hecho pone de manifiesto una vez más el caáacter heterogéneo de RAP. El balanceo que mejor resultado produce es el guiado, con un *speedup* de aproximadamente 1,85. El motivo de que no se alcance el máximo teórico para 2 GPUs, que es 2, es, de nuevo, la heterogeneidad de la aplicación. Este también es el motivo de que, en este caso, el *speedup* alcanzado utilizando sólo 2 GPUs sea superior al obtenido utilizando todo el sistema.

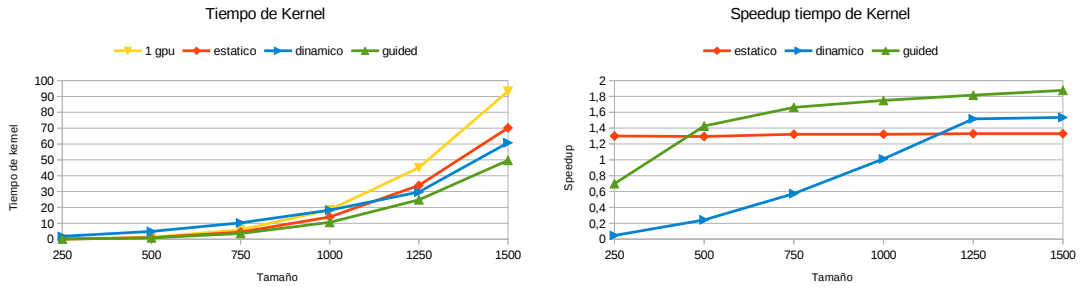

FIGURA 5.10: Comparativa del tiempo de *kernel* en RAP con 2 GPUs

Capítulo 6

Conclusiones

Una vez obtenidos los resultados de las pruebas experimentales utilizando Maat y realizado el análisis pertinente, es el momento de extraer conclusiones a partir del trabajo realizado. Por otra parte, también es el momento de verificar el grado de cumplimiento de los objetivos planteados en el primer capítulo del presente documento. La finalidad de este capítulo es cumplir con este doble propósito y, además, presentar el trabajo futuro relacionado con el tema abordado en este proyecto.

6.1. Implementación del equilibrio de carga

Recordemos que el primer objetivo que se planteó en la sección 1.3 fue: *Implementar el equilibrio de carga de trabajo sobre sistemas heterogéneos, ofreciendo diversas técnicas de balanceo y aprovechando toda la potencia de cómputo disponible.* A este respecto, como se ha explicado detalladamente en el capítulo 4, se ha desarrollado una librería de equilibrio de carga que permite trabajar con tres modelos de distribución diferentes: estático, dinámico y guiado. Además, para la distribución guiada, se ha desarrollado una versión básica homogénea y otra heterogénea. Estos desarrollos dotan de gran versatilidad a la librería y permiten que el usuario, en función de su experiencia y conocimientos, seleccione el que encuentre más adecuado en cada caso.

Considerando los resultados obtenidos a partir del análisis experimental, la conclusión es clara: Maat es capaz de distribuir la carga entre los recursos disponibles, obteniendo una mejora de rendimiento muy considerable independientemente de si el sistema es heterogéneo u homogéneo. Sin embargo, la mejora no es idéntica en ambos tipos de sistemas, luego conviene precisar las conclusiones obtenidas a partir de los resultados para cada uno.

Las pruebas demuestran que, para sistemas heterogéneos y aplicaciones homogéneas, como LavaMD, Maat permite obtener *speedups* cercanos al máximo teórico, aprovechando todos los recursos de los que dispone el sistema. Por otra parte, en el mismo sistema, para aplicaciones heterogéneas (RAP), el *speedup* obtenido, siendo bueno, no se aproxima tanto al máximo teórico como en el caso anterior. Esto sucede por la gran disparidad de potencia de cómputo entre GPUs y CPUs, que provoca que, en caso de asignarse porciones de trabajo grandes a los dispositivos más lentos especialmente cuando se aproxima el final de la ejecución, el rendimiento no sea tan bueno como se esperaba.

Por otro lado, en sistemas homogéneos, al eliminarse las diferencias de potencia de cómputo antes resaltadas, se han obtenido mejoras de rendimiento sustanciales independientemente de si la aplicación es homogénea u heterogénea. Así, tanto en sistemas multiCPU como multiGPU se han obtenido *speedups* cercanos a los máximos teóricos para ambos tipos de aplicaciones. En sistemas multiGPU ejecutando RAP, cabe destacar que, al eliminar las diferencias de potencia y considerar sólo las GPUs del sistema, el *speeedup* obtenido es superior al del sistema heterogéneo; no así para LavaMD. La causa de esto es que la aportación de las CPUs es muy pequeña. Asimismo, se ha observado que, en sistemas multiGPU, ejecutando problemas homogéneos, la diferencia de rendimiento entre las técnicas de balanceo es insignificante, no así si la aplicación es heterogénea o se consideran otros dispositivos. Finalmente, con respecto a sistemas multiCPU, Maat consigue un *speedup* de 2 sobre el reparto de trabajo que realiza OpenCL de forma interna independientemente del tipo de aplicación. Este es un gran resultado, pues permite mejorar la eficiencia de uso de las CPUs de un 50 % a un 100 %.

Si atendemos a cada una de las técnicas de equilibrio utilizadas, la que mejores resultados obtiene es el **balanceo guiado** (heterogéneo o básico según corresponda, de acuerdo con el sistema y aplicación estudiadas). El motivo de que esta sea la mejor técnica, es que es capaz de mantener un carácter dinámico, que le permite adaptarse a las variaciones de carga, al tiempo que disminuye la cantidad de puntos de sincronización con respecto de un balanceo dinámico. Además, este método distribuye la carga de manera fina hacia el final de la ejecución, que es cuando resulta necesario. Por su parte, el **balanceo estático**, por minimizar el número de puntos de sincronización, es el que mejor se comporta para tamaños de problema pequeños. Además, esta técnica genera un *speedup* aproximadamente constante con independencia del tamaño de problema. Finalmente, el **balanceo dinámico** es el que peor se comporta para tamaños de problema pequeños, debido a su exceso de puntos de sincronización. En problemas homogéneos, este método genera un rendimiento similar al estático, pero tiene la ventaja de no requerir que se calculen las potencias de los dispositivos. Este es un parámetro que es necesario determinar a priori y que depende fuertemente de cada aplicación, por lo que puede que en ocasiones no esté disponible. En problemas heterogéneos la mejora de rendimiento de este método

es bastante limitada. La causa es que, pese a que la planificación sea dinámica, esta no considera la potencia de los dispositivos, luego, si una porción de trabajo grande es asignada a un dispositivo lento, el resultado será que este dispositivo ralentice la terminación de toda la aplicación. Sin embargo, cabe destacar que el *speedup* en sistemas multiCPU antes destacado se obtuvo utilizando equilibrio dinámico.

6.2. Permitir el desarrollo de código portable

Uno de los principales problemas de OpenCL, en lo que a manejo de sistemas heterogéneos se refiere, es que gestiona los recursos (plataformas, contextos, dispositivos, *buffers* de memoria...) de forma independiente. Esto, hace que cualquier cambio en el *hardware* o migración de una aplicación a un sistema nuevo, implique una modificación del código para gestionar los nuevos dispositivos. Maat consigue, mediante sus estructuras de datos y funciones, ofrecer una **visión única del sistema**, de modo que la gestión de las estructuras de datos que OpenCL requiere, se realice internamente y de forma transparente al usuario. Las funciones de Maat se encargan de garantizar la adecuada inicialización y buen aprovechamiento, al utilizar las técnicas de balanceo de carga implementadas, de todo el *hardware* disponible, independientemente de si el sistema es heterogéneo con varias CPUs y GPUs o si sólo desean aprovecharse las GPUs de forma homogénea. Con todo esto, las funciones trabajan con el sistema como un conjunto, de manera que la arquitectura subyacente resulta transparente al programador. Con todo esto, la portabilidad de las aplicaciones desarrolladas utilizando Maat queda garantizada, pudiendo migrar una aplicación entre sistemas o añadir/eliminar *hardware* sin necesidad de modificar el código.

6.3. Simplificación de la labor de programación

La gestión individual de las estructuras de OpenCL, aparte de limitar la portabilidad del código, también provoca que este sea difícil de mantener y desarrollar. Esto ocurre porque las tareas asociadas a la inicialización y gestión del entorno, además del manejo de las estructuras de memoria, que es dependiente de los dispositivos, son responsabilidad total del programador y suponen una sobrecarga de trabajo muy grande. Esto se deriva de la generalidad de OpenCL, que es un lenguaje pensado para trabajar con múltiples dispositivos muy variados entre sí. Maat, con su visión única del sistema, descarga al programador de estas responsabilidades, al permitir, mediante sus funciones, que este se relacione con el sistema completo, que es, además, una forma de trabajo mucho mas natural. Por esto, la utilización de Maat resulta en una gran disminución de la cantidad de

líneas de código necesarias para gestionar un sistema, como puede apreciarse de acuerdo a los datos contenidos en la tabla 4.1. Además, debido a la visión única del sistema ya mencionada, el número de líneas necesarias para lanzar una aplicación, a diferencia de al utilizar OpenCL estándar, será independiente del número de dispositivos presentes en el sistema. La causa de esto es que, con Maat, por ejemplo, sólo es necesario llamar una vez a la función que inicializa dispositivos pues, de forma lógica, el sistema completo se comporta como un único dispositivo, o, por el mismo motivo, sólo es necesario llamar una vez a la función de escritura de *buffers* para pasar un *array* a los dispositivos. En definitiva, la utilización de Maat redunda en una simplificación de las labores de inicialización y gestión de dispositivos, así como del lanzamiento de tareas y el manejo de memoria. Esto provoca una disminución de la longitud del código y del número de errores, haciendo que este sea más fácilmente mantenible.

6.4. Trabajo futuro

Una vez cumplidos los objetivos del proyecto, existen algunos aspectos de importancia que han aparecido a raíz del trabajo desarrollado y que merecen un análisis futuro independiente. A saber, estos aspectos son:

- Tratar de eliminar o, al menos, disminuir, la sobrecarga observada en relación con el tiempo de inicialización del *driver*. La solución a este problema podría pasar por mantener el *driver* arrancado, de modo que se ahorre el tiempo de inicialización. La idea que se ha planteado es diseñar un sistema cliente- servidor, de forma que el servidor mantenga arrancados los *drivers*.
- Disminuir el impacto del balanceo de carga sobre el tiempo de comunicación. La técnica planteada para intentar alcanzar este objetivo, es tratar de escribir en cada dispositivo exclusivamente las porciones de datos que vaya a necesitar a lo largo de la ejecución del *kernel*.
- Conseguir que la librería sea capaz de escoger autónomamente los parámetros de balanceo, alcanzando con ello rendimientos cercanos a los óptimos que se obtendrían al fijarlos manualmente. Esta es una tarea bastante complicada sobre la que existe bastante bibliografía disponible.

Bibliografía

[1] Top500. Top 500 supercomputer sites. http://www.top500.org/, 2014. Accedido Junio de 2014.

[2] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[3] NVIDIA. The CUDA homepage. http://www.nvidia.com/object/cuda_home.html, 2007. Accedido Junio de 2014.

[4] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[5] Cedric Augonnet, Jerome Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, ICPADS '10, pages 291–298, Washington, DC, USA, 2010. IEEE Computer Society.

[6] Siegfried Benkner, Sabri Pllana, Jesper Larsson Träff, Philippas Tsigas, Uwe Dolinsky, Cédric Augonnet, Beverly Bachmayer, Christoph W. Kessler, David Moloney, and Vitaly Osipov. Peppher: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 31(5):28–41, 2011.

[7] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In Calin Cascaval and Pen-Chung Yew, editors, *Principles and Practice of Parallel Programming PPOPP*, pages 277–288. ACM, 2011.

[8] Carlos S. de La Lama, Pablo Toharia, José Luis Bosque, and Oscar David Robles. Static multi-device load balancing for OpenCL. In *International Symposium on Parallel and Distributed Processing with Applications ISPA*, pages 675–682. IEEE, 2012.

[9] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. Load balancing in a changing world: Dealing with heterogeneity and performance variability. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 21:1–21:10, New York, NY, USA, 2013. ACM.

[10] Alejandro Acosta, Robert Corujo, Vicente Blanco, and Francisco Almeida. Dynamic load balancing on heterogeneous multicore/multiGPU systems. In Waleed W. Smari and John P. McIntire, editors, *HPCS*, pages 467–476. IEEE, 2010.

[11] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[12] AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK) . http://developer.amd.com/sdks/amdappsdk/. Accedido Junio de 2014.

[13] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[15] Toshihide Ibaraki and Naoki Katoh. *Resource Allocation Problems: Algorithmic Approaches*. MIT Press, Cambridge, MA, USA, 1988.