Universidad Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

Departamento de Ciencias de la Computación

INGENIERÍA INFORMÁTICA

CURSO ACADÉMICO 2009/2010

JavaOpenCL

Binding Java para OpenCL

Trabajo Fin de Carrera

– Autor –

Jesús Sánchez-Oro Calvo

–Tutores–

Raúl Cabido Valladolid

Micael Gallego Carrillo

19 de mayo de 2010

Agradecimientos

En primer lugar debo agradecer a Raúl, Mica y Patxi la confianza depositada en mi al ofrecerme realizar este proyecto. Y agradecerles también la disponibilidad que han tenido a lo largo del proyecto y la atención prestada, así como el continuo apoyo recibido en todo momento. He aprendido más este año con vosotros que en varios años de carrera.

A Leticia, por haber estado apoyándome, animándome a seguir y ayudándome en todos los problemas que han surgido desde el inicio de la carrera hasta la finalización de este proyecto, tanto los relacionados con la universidad como con el exterior. Si no hubieras estado, no sé si habría sido capaz de continuar con la carrera en algunos momentos.

A mi familia por todo el apoyo recibido en las decisiones tomadas a lo largo del proyecto, y por facilitarme todos los beneficios que han estado en su mano para simplificar en la medida de lo posible las dificultades encontradas.

A todos los compañeros que han estado ahí, y en especial a David, por haberme demostrado ser mejor compañero y amigo en este último año de laboratorio que muchos otros a lo largo de toda la carrera.

Gracias a todos por haber estado ahí en todo momento.

Resumen

En los últimos años, el diseño de las arquitecturas de consumo se ha centrado en el paralelismo como objetivo principal para el incremento del rendimiento. Por ello, las mejoras clásicas como el aumento de la frecuencia de reloj se han quedado atrás en favor de la inclusión de un mayor número de procesadores en la CPU. Por otro lado, impulsadas por la industria de lo videojuegos, las GPUs han evolucionado hacia unidades de cómputo altamente paralelas, programables y con un elevado ancho de banda en memoria. Debido a que en la actualidad la gran mayoría de los sistemas de cómputo incluyen CPUs, GPUs y otros tipos de procesadores, es necesario disponer de software que sea capaz de aprovechar el poder de cómputo presente en estas arquitecturas heterogéneas.

OpenCL es un estándar multiplataforma para la computación en sistemas heterogéneos que surge en diciembre de 2008 que ha ido adquiriendo cada vez más importancia. Esto se debe principalmente a su eficiencia, así como su compatibilidad con la gran mayoría de dispositivos destinados a la programación paralela. Además, OpenCL está respaldado por el consorcio de empresas que conforman el grupo Khronos, entre las que se encuentran compañías como NVIDIA, AMD o Apple, lo que impulsa aún más su expansión.

Sin embargo, OpenCL ha sido desarrollado para ser utilizado desde C/C++, lo que limita su utilización por parte de desarrolladores no acostumbrados a programar en estos lenguajes. Por otra parte, el lenguaje de programación Java se encuentra entre los lenguajes más utilizados, por lo que resulta de gran interés acercar a los programadores de Java a tecnologías como OpenCL sin necesidad de cambiar el lenguaje de programación utilizado. Ésta es la motivación principal de

este proyecto, crear una API que recubra la especificación oficial de OpenCL con el lenguaje Java, de manera que cualquier programador de Java pueda programar arquitecturas heterogéneas sin necesidad de conocer lenguajes como C/C++.

Índice general

Índice de figuras

Capítulo 1

Introducción

En los últimos años, las arquitecturas de consumo están cada vez más orientadas a aprovechar el paralelismo para incrementar su rendimiento. Esto es debido principalmente a lo que se denomina *Performance Wall* (ver [6]). Este término se refiere al límite alcanzado por los procesadores utilizando las técnicas tradicionales de aumento de rendimiento, como es el aumento de la frecuencia de reloj del procesador. El problema de estas técnicas aparece cuando se alcanzan los límites físicos para el aumento de las prestaciones. Un vez superado dicho límite, es necesario utilizar otra clase de mejoras que permitan continuar con el incremento del rendimiento de estos dispositivos. Por este motivo, en los últimos años la tendencia seguida ha pasado del incremento en la frecuencia de reloj de un procesador, a la inclusión de varios procesadores, restando importancia a la frecuencia de éstos. La evolución seguida por las GPU ha sido completamente distinta. Las GPUs surgieron como dispositivos de cómputo paralelo, pero destinados a realizar procesos específicos como el renderizado de gráficos. Sin embargo, en los últimos años han evolucionado hasta convertirse en procesadores paralelos programables, orientados hacia la programación de propósito general. Por este motivo surge la necesidad de herramientas capaces de explotar el poder de cómputo de estas plataformas heterogéneas.

Crear aplicaciones para plataformas heterogéneas no es sencillo, debido a que los modelos de programación tradicional y aquellos orientados al desarrollo sobre

plataformas *multi-core* y *many-core* son muy diferentes. Los modelos tradicionales se basan normalmente en estándares que asumen un espacio de memoria común y no abarcan de forma explícita las operaciones vectoriales. Sin embargo, los modelos de programación de propósito general sobre GPU añaden jerarquías complejas de memoria y operaciones vectoriales, pero son generalmente dependientes del hardware, la plataforma y el fabricante. Estas limitaciones hacen difícil acceder al poder de cómputo de los diferentes procesadores heterogéneos desde un único código fuente multiplataforma. Además, es necesario tener en cuenta que además de CPU y GPU, una arquitectura heterogéneas puede constituirse de otros dispositivos como los DSP (*Digital Signal Processor*) o el procesador Cell.

Para ello, a lo largo de las últimos años han surgido diferentes herramientas que proporcionan al desarrollador la posibilidad de aprovechar el rendimiento de estos nuevos tipos de procesadores. Entre estas herramientas destacan las orientadas al cómputo sobre GPUs, como pueden ser CUDA, de NVIDIA, y sobre CPUs con varios procesadores, como son OpenMP o Ct (Intel). Todas estas herramientas comparten la misma limitación, ser compatibles tan sólo sobre un hardware determinado.

Debido al aumento del interés en este área, aparece la necesidad de una herramienta multiplataforma, independiente del fabricante del dispositivo, y que no sólo sea capaz de aprovechar el rendimiento de los dispositivos gráficos, sino también de los procesadores multinúcleo y otros tipos de dispositivos de cómputo, como pueden ser los procesadores embebidos (por ejemplo, DSP). Así, surge OpenCL, un estándar para la programación de propósito general desarrollado por Khronos Group.

Khronos Group es un consorcio industrial que tiene como objetivo el desarrollo de estándares libres centrados en la computación paralela y el procesamiento de gráficos sobre todo tipo de plataformas. Entre estos estándares se encuentran OpenCL, OpenGL, WebGl, etc. En la Figura 1.1 aparecen los principales componentes de dicho grupo.

Figura 1.1: Miembros del grupo Khronos

Desde su salida, OpenCL ha ido desarrollándose hasta convertirse en el primer estándar de código abierto para la programación de propósito general sobre arquitecturas heterogéneas, proporcionando a los desarrolladores software un acceso eficiente e independiente de la plataforma a estos nuevos dispositivos.

Uno de los objetivos de OpenCL es que todos los recursos computacionales del sistema puedan aprovecharse para realizar cómputo de propósito general. OpenCL se ha desarrollado para utilizarse con C/C++, un lenguaje que proporciona un mayor rendimiento, con la penalización de que el código sea dependiente del compilador. De esta limitación surge la necesidad de hacer que OpenCL sea, en la medida de lo posible, un lenguaje portable a todos los dispositivos sin necesidad de modificaciones en el código ni de diferentes compilaciones, pero sin perder su eficiencia. Con esto, un programador sería capaz de desarrollar código que pueda ser compilado en una sóla máquina, pero ejecutado en cualquier sistema operativo utilizando diferentes tipos de dispositivos (CPUs, GPUs, etc.), aumentando el alcance original de la herramienta.

Posición en abril de 2010	Posición en Abril de 2009	Lenguaje de Programación
1	2	C
2	1	Java
3	3	C++
4	4	PHP
5	5	(Visual) Basic

Cuadro 1.1: Lista *TIOBE* del mes de Abril de 2010. Esta lista presenta los lenguajes de programación más utilizados, donde Java ocupa un segundo puesto por detrás de C.

Por este motivo, este proyecto presenta JavaOpenCL, una API que otorga al desarrollador la posibilidad de utilizar OpenCL en Java, uno de los lenguajes de programación más comunes (ver tabla 1.1). La unión de Java y OpenCL proporciona todas las ventajas de ambos lenguajes, como pueden ser la eficiencia de OpenCL unido a la portabilidad de Java, así como su tratamiento de los errores. Además, gracias a Java, se proporciona al desarrollador una interfaz más simple que la utilizada por OpenCL, facilitando su aprendizaje y posterior uso, y evitando en gran medida los problemas comunes que puedan surgir a un programador inexperto en C (gestión de memoria, punteros, etc.).

1.1. Computación paralela

La velocidad de los computadores secuenciales convencionales se ha incrementado continuamente para adaptarse a las necesidades de las aplicaciones, hasta llegar a encontrarse con los límites físicos (*Performance Wall*). Pero en diversas áreas sigue siendo necesario un poder computacional superior, como el modelado y solución numérica de problemas en ciencias e ingeniería, o los costosos cálculos iterativos sobre grandes cantidades de datos con fuertes restricciones temporales. Estos sistemas se vuelven cada vez más complejos requiriendo una mayor capacidad de cómputo. Pero esto no siempre es posible debido a las limitaciones físicas

que impone el desarrollo de procesadores.

Para hacer frente a estas limitaciones se ha optado por la utilización de varios procesadores conformando un sistema paralelo. El sistema paralelo proporciona un gran abanico de opciones para aumentar el rendimiento, entre las que se encuentran la utilización de un *pipeline*, el paralelismo a nivel de instrucción, la ejecución fuera de orden o la especulación, entre otras.

La programación paralela se basa en la utilización de varios procesadores de manera conjunta para resolver una tarea común. La manera en la que cada procesador va a afrontar el problema es definida por el programador, de forma que cada procesador trabaja sobre una porción del problema, intercambiando los resultados que sean necesarios a través de memoria compartida o con el uso de una red de interconexión.

La computación paralela permite, entre otras cosas, resolver problemas que de otra manera serían inabordables, ya sea por capacidad de cómputo, o por el tiempo empleado en resolverlo. En este proyecto existen dos niveles de paralelismo, los cuales se muestran a continuación:

- **Programación concurrente:** Varios procesos trabajando en la solución de un problema, puede ser paralela (con varios procesadores)

- **Computación heterogénea:** Varios procesadores de diferentes características trabajando en la solución de un mismo problema.

La programación concurrente está presente en todo momento, ya que independientemente del dispositivo utilizado para solucionar un problema, va a disponer de varios núcleos, los cuales van a trabajar sobre un espacio de memoria compartida en un mismo instante de tiempo.

La computación heterogénea es otra de las bases de este proyecto, ya que es posible utilizar diferentes dispositivos para resolver un mismo problema, distribuyendo la carga de trabajo entre ellos de manera que el problema sea eficiente.

1.2. OpenCL

OpenCL [4] (*Open Computing Language*) es el nombre que recibió el estándar de programación sobre arquitecturas paralelas desarrollado y liberado por Khronos. Está respaldado por las principales compañías que producen hardware y software relacionado con la computación paralela, como son AMD, NVIDIA, Apple, IBM, Intel, etc. Esta tecnología está empezando a cobrar gran importancia en el mundo de la computación de propósito general sobre GPUs.

Además, al tratarse de un estándar reconocido, ya no es necesario aprender un lenguaje para programar sobre tarjetas de una compañía concreta y otro completamente diferente para programar sobre tarjetas una compañía diferente, si no tan sólo es necesario disponer de *drivers* compatibles y de las librerías que permiten el desarrollo en OpenCL. Esto ha llevado a OpenCL a un incremento notable en su uso desde su lanzamiento. En algunos casos, como en el nuevo Mac OS X Snow Leopard, no es necesaria la instalación por parte del usuario de ninguna herramienta especial, ya que se encuentra integrada con el sistema operativo.

Por último, lo que ha conseguido que OpenCL siga cobrando más importancia es su portabilidad. Es importante tener en cuenta que dicha portabilidad es sólo funcional. Esto es debido a que aunque los resultados obtenidos de la aplicación sean correctos en diferentes dispositivos, para obtener el mejor rendimiento es necesario optimizar el código para su utilización en un dispositivo concreto. Por este motivo, una misma aplicación, aún funcionando en dos dispositivos diferentes, no obtendrá el mismo rendimiento en ambos.

1.3. Programación de propósito general en GPU (GPGPU)

Las diferentes SDK disponibles durante la realización de este proyecto tan sólo ofrecen soporte para la programación de OpenCL sobre dispositivos gráficos (GPU), no sobre procesadores comunes (CPU), por lo que el principal objetivo de

este proyecto se ha centrado sobre las GPUs, al carecer de soporte sobre CPU. Por este motivo, este proyecto se ha centrado en la programación de propósito general en sobre GPU (GPGPU).

GPGPU son las siglas de *General-Purpose computation on Graphics Processing Units*, es decir, computación de propósito general sobre unidades de proceso gráfico (GPU). Las GPU son procesadores de alto rendimiento formados por múltiples núcleos capaces de llevar a cabo grandes operaciones sobre diversos datos con un gran rendimiento.

Aunque en sus comienzos las GPU estaban orientadas principalmente a los gráficos y eran muy difíciles de programar, en la actualidad se han convertido en procesadores de propósito general paralelos que soportan interfaces de alto nivel que permiten su programación bajo lenguajes como C/C++.

La GPU se muestran como una plataforma adecuada para la ejecución de tareas que puedan expresarse en forma de cómputo paralelo de datos, lo que la convierte en un dispositivo muy eficiente para aquellos problemas que puedan ser paralelizables, perdiendo toda esta efectividad frente a problemas secuenciales.

Capítulo 2

Estado del arte

Al comienzo de este proyecto no existía ninguna implementación de OpenCL para Java, pero durante la realización del mismo han ido surgiendo diversas implementaciones, aunque ninguna de ellas recubre OpenCL de manera completa, y con suficientes ejemplos probados para demostrar su funcionamiento. Las implementaciones encontradas se detallan a continuación.

- **NativeLibs4Java [7]:** Se trata de un *binding* desarrollado con JNA, y mantiene una interfaz complicada debido a los siguientes problemas:
  - El generador automático de JNA crea varias opciones para cada función recubierta. Es difícil para el programador decidir cuál es la correcta en cada caso.
  - Es necesario que el usuario conozca JNA para utilizarlo, ya que utiliza clases propias de JNA para ser utilizado.
  - Existen pocos ejemplos disponibles.
- **jocl(1) [2]:** Se trata del *binding* más estable de los actuales. Su problema principal reside en que *jocl* es un *binging* directo, por lo que no sigue la orientación a objetos de Java, ni respeta los convenios de programación Java. Es libre tan sólo para proyectos que no sean comerciales, lo que limita su uso.

- **jocl(2) [11]:** Presenta el mismo nombre que el anterior, y ha sido desarrollado por los creadores de JOGL (implementación de Java para OpenGL) y JOAL (implementación de java para OpenAL). El código JNI se ha generado automáticamente con GlueGen. Al estar desarrollado el código JNI automáticamente, se complica su depuración.

- **opencl4j: [7]** Muy similar a NativeLibs4Java pero con la diferencia de que éste posee una interfaz menos compleja para el usuario.

Todos estos *bindings* tienen una escasa documentación y muy poca cantidad de ejemplos que demuestren su funcionamiento (apenas dos o tres en los casos más avanzados).

Por otro lado, es importante destacar la existencia de diversos *bindings* de OpenCL para otros lenguajes de programación, entre los que se incluyen los mostrados a continuación.

- **Python::OpenCL [5]:** Se encuentra bajo licencia GPL. Permite que el código OpenCL se implemente de manera directa en el código Python. Para ello se necesita incluir el código dentro de un objeto de la clase `Program` implementada por Python::OpenCL, para después poder ejecutar dicho código como si de una función de Python se tratase. Es limitado, ya que es el propio *binding* el que se encarga de reservar la memoria en el dispositivo (CPU, GPU, DSP, etc.) y prepararlo para la ejecución, lo que resta libertad al programador.

- **Open Toolkit Library [3]:** Es una librería de bajo nivel desarrollada en C# que recubre OpenGL, OpenCL y OpenAL. Puede ser utilizada en cualquier lenguaje que soporte Mono o .NET, como son C# y VB.Net.

Capítulo 3

Objetivo

3.1. Objetivo general

El objetivo principal de este Proyecto es la generación de una API para Java que permita la programación de arquitecturas de cómputo heterogéneas bajo el estándar OpenCL. Además, esta API será multiplataforma, de manera que los usuarios de los principales sistemas operativos dispongan de las herramientas necesarias para programar OpenCL desde Java.

Además, esta API deberá ser fiel a la original, a fin de que la curva de aprendizaje sea lo más sencilla posible, para facilitar su utilización. Esta fidelidad está referida sólo a la interfaz de la API, ya que se aprovecharán las ventajas proporcionadas por Java de manera que la tarea del programador se simplifique.

3.2. Objetivos parciales

Los objetivos parciales de este proyecto se derivan del objetivo principal y se enumeran a continuación:

- Realizar un estudio en profundidad del estándar OpenCL así como la especificación del mismo. Para ello se utilizará la documentación disponible en la web de Khronos, además de diferentes documentos donde se exponen técnicas de programación de altas prestaciones.
- Estudio del desarrollo de APIs para el lenguaje Java, mediante el estudio de otras APIs disponibles.
- Estudio de los posibles mecanismos de comunicación entre C y Java, y elección de uno de ellos. Entre los disponibles, se eligen para el estudio JNI y JNA, por ser los más extendidos.
- Desarrollo de la API JavaOpenCL que recubrirá la especificación original de OpenCL para el lenguaje Java.
- Comprobación del correcto funcionamiento de la API, para lo cuál se portará un conjunto de ejemplos disponibles en la SDK de OpenCL compuesta de pruebas de rendimiento y de aplicaciones gráficas, abarcando la especificación completa.
- Desarrollo de una web con toda la información relacionada con la API, tanto de su desarrollo como de su uso, así como un manual de instalación y de utilización, incluyendo la documentación JavaDoc (`http://www.gavab.es/wiki/JavaOpenCL`).
- Una vez finalizada la API y probada su funcionalidad, realización de un modelado orientado a objetos de la misma, de manera que se acerque más al concepto de programación Java.

Capítulo 4

Metodología y tecnologías utilizadas

4.1. Metodología

Para la elección de la metodología a utilizar se han tenido en cuenta principalmente dos alternativas: el Proceso Unificado de Desarrollo y las Metodologías Ágiles.

La elección de ambas alternativas se basa en la utilidad que cada una de ellas tiene para este proyecto. Desde una primera aproximación, el Proceso Unificado es útil para tener un control más estricto y mejor documentado de la evolución del proyecto, mientras que la utilización de una metodología ágil se ajusta mejor a los cambios que puedan surgir durante el desarrollo del proyecto, muy común en el desarrollo de este tipo de proyectos, teniendo en cuenta la constante evolución de OpenCL.

4.1.1. Proceso Unificado de Desarrollo

Es una de las metodologías tradicionales más extendidas, y se caracteriza por estar dirigido por casos de uso, centrado en la arquitectura y ser iterativo incremental.

- **Dirigido por casos de uso:** los casos de uso se utilizan para capturar los requisitos funcionales y para definir el contenido de las iteraciones. Es decir, cada iteración escoge un conjunto de casos de uso y desarrolla el camino a través de todas las fases.

- **Centrado en la arquitectura:** se asume que no existe un único modelo que represente todos los aspectos del sistema. Por eso existen diversos modelos y vistas que cubren toda la arquitectura software del sistema.

- **Iterativo e incremental:** se compone de cuatro fases claramente diferenciadas: inicio, elaboración, construcción y transición. Cada una de estas fases se divide en iteraciones, que ofrecerán un incremento al finalizar que añade o mejora funcionalidades del sistema en desarrollo.

4.1.2. Metodologías Ágiles

Las metodologías ágiles tienen un enfoque más práctico y menos dirigido a la documentación que las metodologías tradicionales, como el proceso unificado. Según su manifiesto, se valora:

- **Al individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas:** la gente es el principal factor de éxito de un proyecto software. Por ello, se considera más importante conseguir un buen equipo que construir el entorno. Por ello, se elige primero al equipo y después se deja que sean éstos los que construyan el entorno de desarrollo en base a sus necesidades.

- **Desarrollar software que funciona más que conseguir una buena documentación:** sigue la regla de no producir documentos a no ser que sean necesarios de forma inmediata para tomar una decisión importante. Además, los documentos generados deben ser cortos y centrarse en lo fundamental.

- **La colaboración con el cliente más que la negociación de un contrato:** se propone una interacción constante entre el cliente y el equipo de

desarrollo. Esta colaboración será la que marque la marcha del proyecto y asegure su éxito.

- **Responder a los cambios más que seguir estrictamente un plan:** la habilidad de responder a los cambios que puedan surgir a lo largo del proyecto también determinan el éxito del mismo. Por ello, la planificación debe ser flexible y abierta.

4.1.3. Comparativa de metodologías

La tabla 4.1 muestra una comparativa de ambas metodologías, que ayuda a la elección de la que más se adapta al proyecto actual.

4.1.4. Elección de la metodología

JavaOpenCL es un proyecto en el cual la documentación surge según va siendo desarrollado, además de que conlleva cambios continuos, principalmente por la continua actualización a la que se ve sometida el estándar, así como de nuevas versiones de los diferentes SDK de los principales desarrolladores. Debido a esto, la metodología que más se ajusta al desarrollo de este proyecto es la de las metodologías ágiles, por su mejor adaptación a los cambios que puedan surgir, así como la imposibilidad de adaptación del proyecto a una planificación estricta como la que nos puede ofrecer el Proceso Unificado. Además, dentro de las metodologías ágiles, el desarrollo a seguir será iterativo incremental con un modelo en espiral con prototipos. De esta forma, durante cada iteración se irá incorporando cierta funcionalidad al proyecto a la vez que se corrigen los posibles errores que vayan surgiendo durante el desarrollo, ya que de otra manera sería imposible subsanar los errores con la API completa.

La idea principal es generar un prototipo con la que incluyan las funciones básicas de OpenCL, para luego ir aumentando la funcionalidad hasta conseguir cubrir toda la especificación. Tras cada iteración se realizarán pruebas para comprobar el correcto funcionamiento, mediante la utilización de los ejemplos de OpenCL

Metodologías Ágiles	**Metodologías Tradicionales**
Basadas en heurísticas provenientes de prácticas de producción de código.	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo
Especialmente preparadas para cambios durante el proyecto.	Cierta resistencia a los cambios.
Impuestas internamente (por el equipo).	Impuestas externamente.
Proceso menos controlado, con menos principios.	Proceso mucho más controlado, con numerosas políticas / normas.
No existe contrato tradicional o al menos es bastante flexible.	Existe un contrato prefijado.
El cliente es parte del equipo de desarrollo.	El cliente interactúa con el equipo de desarrollo mediante reuniones.
Grupos pequeños trabajando en el mismo sitio.	Grupos grandes posiblemente distribuidos.
Pocos artefactos y roles.	Muchos artefactos y roles.
Menos énfasis en la arquitectura del software.	La arquitectura del software es esencial y se expresa mediante modelos.

Cuadro 4.1: Comparativa entre metodologías tradicionales y ágiles

portados a JavaOpenCL. Esto asegura que al finalizar el proyecto se dispondrá de una librería con un funcionamiento correcto y con ejemplos que lo demuestran.

La metodología elegida consiste en una serie de ciclos que se repiten en forma de espiral, comenzando por el centro. Cada uno de estos ciclos se compone de las siguientes actividades:

1. **Determinar o fijar objetivos:** Se definen los requisitos de forma detallada, además de las posibles restricciones y los productos a obtener. Se identifican los riesgos del proyecto y las alternativas para evitarlos.

2. **Análisis de riesgos:** Se estudian todos los riesgos potenciales y se seleccionan una o varias alternativas propuestas para reducirlos o evitarlos.

3. **Desarrollar, verificar y probar:** Se desarrollan las funcionalidades del proyecto especificadas en los objetivos fijados. Además, se verifican estas funcionalidades y se desarrollan pruebas para corregir posibles fallos.

4. **Planificar:** Se revisan los resultados obtenidos , evaluándolos, y decidiendo si se continúa con la siguiente fase y planificándola.

La Figura 4.1 representa el esquema que sigue el modelo en espiral.

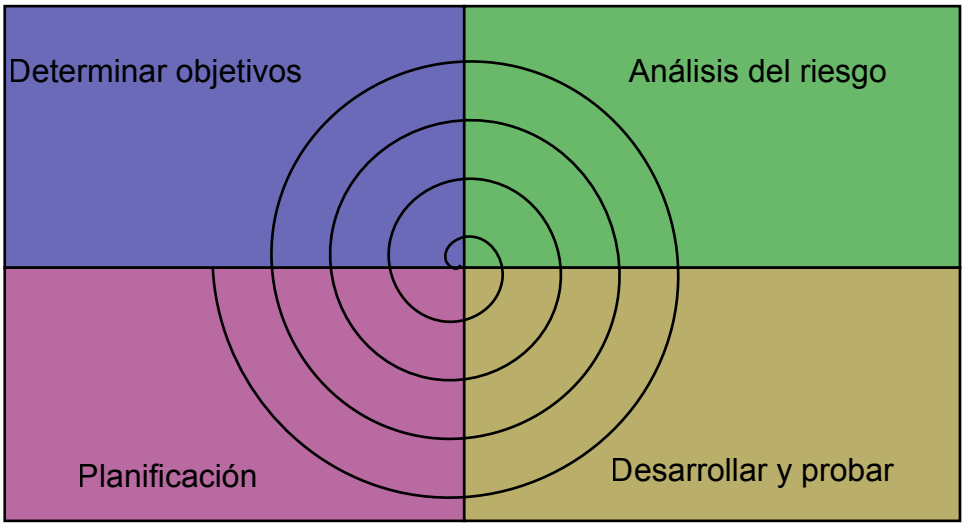

Figura 4.1: Esquema modelo en espiral

4.2. Tecnologías

Las tecnologías utilizadas durante la realización de este proyecto así como su utilización en el mismo se resumen a continuación:

- **Java:** Lenguaje de programación en el cual se ha implementado la API JavaOpenCL.

- **Java Native Interface (JNI):** Se utiliza para poder realizar a llamadas de la API original desde JavaOpenCL.
- **GCC:** Compilador utilizado para generar a librería dinámica.
- **Eclipse:** Entorno de desarrollo de JavaOpenCL.
- **JOGL:** API de *Java* utilizada para renderizar los resultados de los ejemplos gráficos de JavaOpenCL.
- **OpenCL SDK:** API original de OpenCL que contiene las funciones que JavaOpenCL recubre en su especificación.

4.2.1. Java

Java es un lenguaje de programación orientada a objetos desarrollado por Sun Microsystems en los años 90. Su sintaxis es similar a la de lenguajes como C/C++ pero abstrayendo al programador de herramientas de bajo nivel como pueden ser el acceso directo a la memoria (reserva y liberación) y el manejo de punteros.

Una aplicación Java se ejecuta sobre una Máquina Virtual de Java (JVM), que se encarga de ejecutar el código generado por la compilación previa de la aplicación. Ese código generado, denominado *bytecode* es el obtenido utilizando el compilador de Java, de manera que cualquier máquina virtual sea capaz de ejecutarlo.

Los aspectos más importantes de Java son:

- **Es orientado a objetos.** Se trata de un paradigma de programación que abstrae las estructuras de datos utilizadas por los programadores en objetos, entidades que se componen de tres partes:
  - *Estado:* Se compone de los atributos, que almacenan la información referente al objeto, que tendrá unos valores concretos.
  - *Comportamiento:* Está definido por los métodos, que representan las operaciones que se pueden realizar sobre los objetos.

    - *Identidad:* Se trata de una propiedad que consigue que cada objeto sea diferente del resto.

- **Independencia de la plataforma.** La filosofía de Java dice que cualquier aplicación escrita en Java puede ser ejecutada en cualquier tipo de plataforma, como dice su lema *"write once, run everywhere"*.

- **Recolector de basura.** En Java no es posible liberar la memoria de los objetos reservados, de ello se encarga el Recolector de Basura (*Garbage Collector* - GC). Este recolector libera la memoria de los objetos cuando ya no quedan referencias a los mismo, señal de que no van a ser utilizados en el resto del código.

4.2.2. Java Native Interface (JNI)

JNI es un mecanismo que permite ejecutar código nativo desde Java y viceversa. Llamamos código nativo a las funciones escritas en un lenguaje como C/C++ que se ejecutan sobre el sistema operativo donde está funcionando la máquina virtual de Java. JNI tiene un interfaz bidireccional que permite a las aplicaciones Java llamar a código nativo a la vez que las funciones de código nativo pueden ejecutar métodos de clases Java. Es decir, JNI nos ofrece dos interfaces:

- **Métodos nativos:** permiten que desde Java se realicen llamadas a funciones implementadas en las librerías nativas. Es el interfaz utilizado en este proyecto.

- **Interfaz de invocación:** permite incrustar una Máquina Virtual de Java en una aplicación nativa. Para ello, la aplicación nativa llama a librerías nativas de la máquina virtual y con el interfaz de invocación ejecuta métodos Java en la máquina virtual.

La estructura de JNI puede resumirse en la Figura 4.2.

Para desarrollar aplicaciones Java que utilicen funciones de librerías en C/C++ siempre se siguen los siguientes pasos generales:

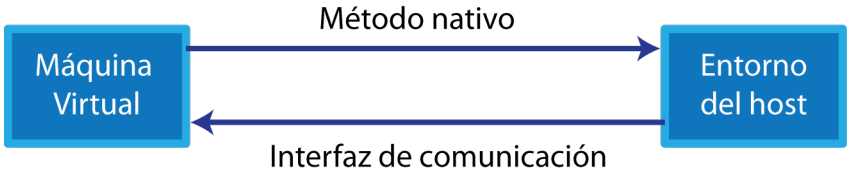

Figura 4.2: Estructura de JNI

1. **Declarar el método nativo como miembro de una clase:** en el código Java, se declaran los métodos que vayan a utilizar las funciones de las librerías nativas con el modificador `native`. Este código debe incluir una llamada a un método para cargar la librería. Este método pertenece a la clase `System`, y su definición es la siguiente:

   ```
   void loadlibrary(String libraryName)
   ```

   Donde `libraryName` es el nombre de la librería a cargar, sin la extensión proporcionada por el sistema operativo (.dll, .so, .dynlib), teniendo en cuenta que se encuentra dentro del PATH o del directorio de trabajo. En caso de no encontrarse en ese directorio, será necesario modificar la variable de entorno PATH para que se incluya el directorio de la librería.

2. **Crear el fichero de cabecera nativo (.h):** el siguiente paso es generar el fichero `.h` con las cabeceras de los métodos nativos que se van a implementar. Esto se puede realizar de una manera sencilla con el comando `javah claseJava`, donde `claseJava` es clase Java compilada en el que hemos declarado los métodos nativos.

3. **Implementar el método nativo:** una vez disponemos del fichero de cabeceras, sólo tenemos que implementar las funciones definidas en el mismo. Para ello, basta con crear un fichero `.c` donde copiaremos las cabeceras del fichero `.h`, e implementaremos las funciones correspondientes. Hay que tener en cuenta si la implementación de las funciones nativas va a ser en C/C++, ya que la declaración de los tipos de JNI es diferente para ambos lenguajes.

4. **Compilar el fichero nativo:** Por último, sólo falta compilar el último fichero para generar la librería dinámica, que servirá las funciones implementadas

para ser llamadas desde Java.

JNI proporciona una serie de funciones y tipos que dan cierta transparencia a la convergencia de tipos entre C y Java. Todas las funciones JNI tienen dos parámetros comunes. El primero de ellos es el parámetro `env`, el cual representa una referencia a una tabla que contiene todas las funciones que pueden ser ejecutadas con JNI. El segundo es el parámetro `jobject`, una referencia al objeto Java que invoca la función JNI. Este último parámetro, en caso de ser invocado por un método estático, se convierte en una referencia a la clase que realiza la invocación.

Pero utilizar JNI conlleva riesgos. El primero de ellos, y quizás el más evidente, es que el código nativo no se va a ejecutar sobre la máquina virtual, por lo que desde el momento que utilizamos JNI debemos tener en cuenta que perdemos la portabilidad que nos ofrece Java. Esta parte será, por tanto, la única dependiente del sistema operativo, de manera que se dispondrá de diferentes librerías a utilizar en función de la plataforma utilizada. Por otro lado, un fallo en la aplicación de código nativo puede hacer que la aplicación deje de funcionar, en apariencia sin motivo alguno. Este último aspecto se controlará en la medida de lo posible mediante la transformación de los posibles fallos en el código nativo en excepciones de Java.

4.2.3. GCC

GCC son las siglas de *GNU Compiler Collection*. Se trata de un sistema de compiladores desarrollado por el *GNU Project* que soporta diferentes lenguajes de programación. Se ha convertido en el compilador estándar en la gran mayoría de sistemas basados en Unix, como pueden ser GNU/Linux o Mac OS X. Además, ha sido portado a una gran variedad de arquitecturas de procesadores, y es muy utilizado en el desarrollo de todo tipo de software. También se encuentra disponible para la gran mayoría de plataformas embebidas, como Symbian.

Aunque en Windows no está disponible la versión original de GCC, se dispone de una adaptación muy utilizada: MinGW. Se trata del compilador de GCC para Windows, además de una serie de librerías de libre distribución para la API de

Windows, permitiendo a los desarrolladores crear aplicaciones nativas para Microsoft Windows.

4.2.4. Eclipse

Eclipse es un entorno de desarrollo abierto que en su versión básica permite desarrollar aplicaciones Java. Su principal ventaja es que por medio de la instalación de *plugins* es posible desarrollar código para otros tipos de lenguajes, como C, Haskell e incluso Latex. Por esto ha sido el entorno elegido para desarrollar JavaOpenCL, porque permite desarrollar el código en C el código en Java de manera simultánea, sin necesidad de cambiar entre entornos de desarrollo.

EclipseGavab 2.0 es una distribución de Eclipse que contiene todo lo necesario (*plugins* y compiladores) para desarrollar software para diversos lenguajes de programación, entre los que se incluyen Java, C/C++, FreePascal, Ruby o Haskell. Además, incluye herramientas destinadas al desarrollo colaborativo, como es Subversive, un cliente de Subversion. Dispone de versiones tanto para Windows como para Ubuntu, los dos sistemas operativos tratados en el proyecto.

4.2.5. JOGL

JOGL es una API de Java diseñada para proporcionar soporte gráfico 3D sobre hardware a las aplicaciones escritas en Java. Mediante JOGL es posible acceder toda la especificación original de OpenGL. Básicamente, JOGL permite que una aplicación Java pueda utilizar una API gráfica como OpenGL para renderizar gráficos igual que lo haría una aplicación nativa. En este proyecto es útil para la visualización de los ejemplos gráficos portados.

4.2.6. OpenCL SDK

OpenCL SDK es un *framework* completo para la programación paralela e incluye un lenguaje, una API y las librerías necesarias para la ejecución de aplicaciones. El modelo de la plataforma de OpenCL consiste en una máquina (*host*) conectada a uno o varios dispositivos que soportan OpenCL. Estos dispositivos se dividen en una o varias unidades de cómputo (*CU - compute units*), las cuáles están divididas en elementos de proceso (*PE - processing elements*). La aplicación OpenCL ejecuta comandos desde el *host* para ejecutar cálculos en los elementos de proceso dentro de un dispositivo. La Figura 4.3 muestra la organización de una plataforma OpenCL.

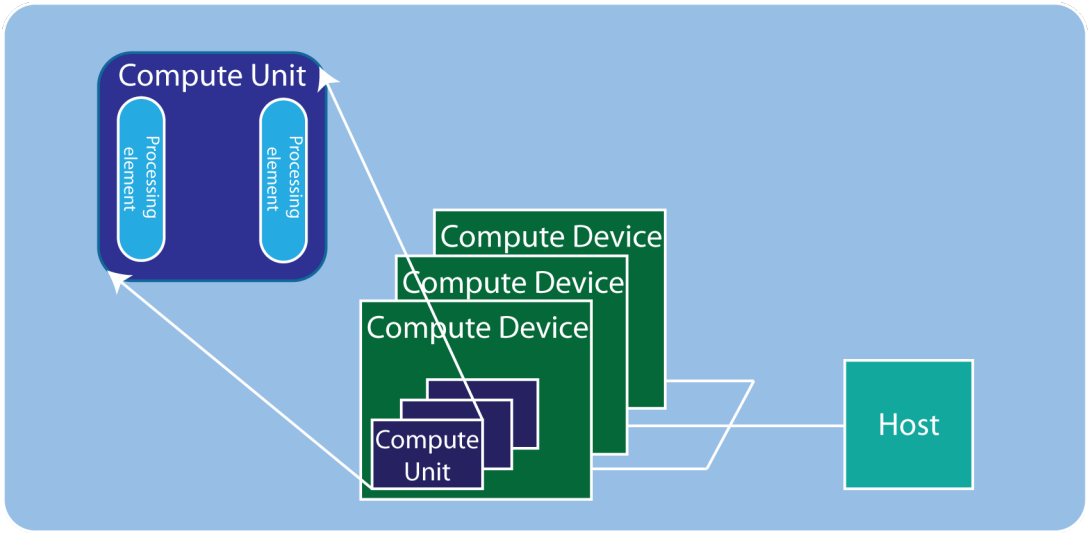

Figura 4.3: Modelo de plataforma de OpenCL

Para entender dicho modelo, es necesario conocer qué es un *kernel* en OpenCL. Un *kernel* es una función que se declara en un programa y se ejecuta en un dispositivo OpenCL, de forma paralela. La Figura 4.4 muestra las diferencias de ejecución de un programa con un sólo hilo de ejecución frente a un programa ejecutando varias instancias de un kernel sobre un mismo conjunto de datos.

El modelo de ejecución de un programa OpenCL se divide en dos grandes partes: los *kernels* que se ejecutan en uno o más dispositivos OpenCL y el programa

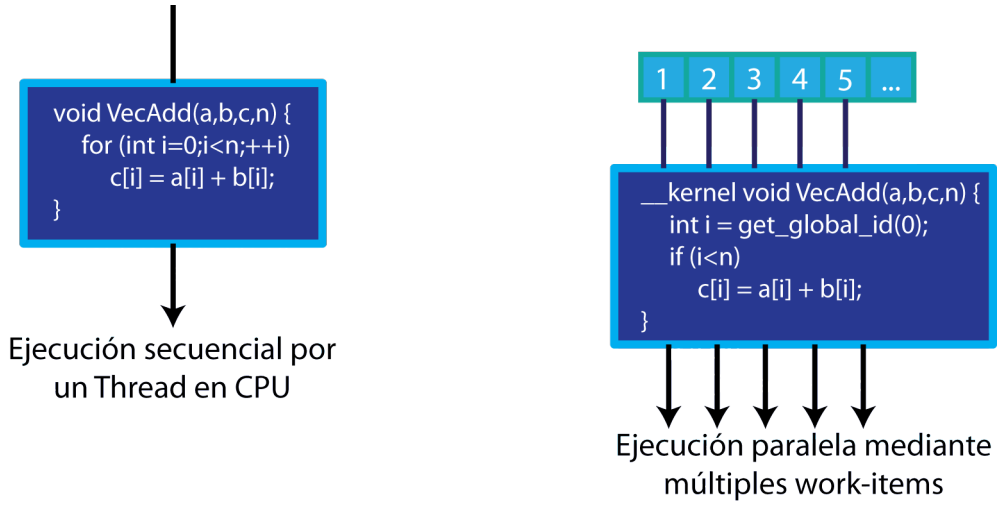

Figura 4.4: Diferencias entre ejecución en un sólo Thread y en múltiples Threads

que se ejecuta en el *host*. Este programa define el contexto para los *kernels* y gestiona su ejecución. El núcleo del modelo de ejecución de OpenCL define como se ejecutan los *kernels*. Cuando la máquina ordena la ejecución de un *kernel*, se define un espacio de índices, llamado *NDRange*. Una instancia del *kernel* se ejecuta para cada punto de este espacio de índices, y se denomina *work-item*. Cada uno de esos *work-items* ejecuta el mismo código pero el camino de ejecución y los datos modificados pueden ser diferentes para cada uno. Estos *work-item* se organizan en *work-groups*, los cuales proveen una descomposición más abstracta del espacio de índices. Los *work-items* dentro de un *work-group* se ejecutan de manera concurrente en los elementos de proceso de una unidad de cómputo. La Figura 4.5 muestra la organización descrita.

La máquina define un contexto para la ejecución de los *kernels*. Este contexto incluye los siguientes recursos:

- **Devices:** Los dispositivos OpenCL que van a ser utilizados por la máquina.
- **Kernels:** Las funciones OpenCL que se van a ejecutar en los dispositivos.
- **Objetos de programa:** El código fuente y ejecutable que implementa los kernels.

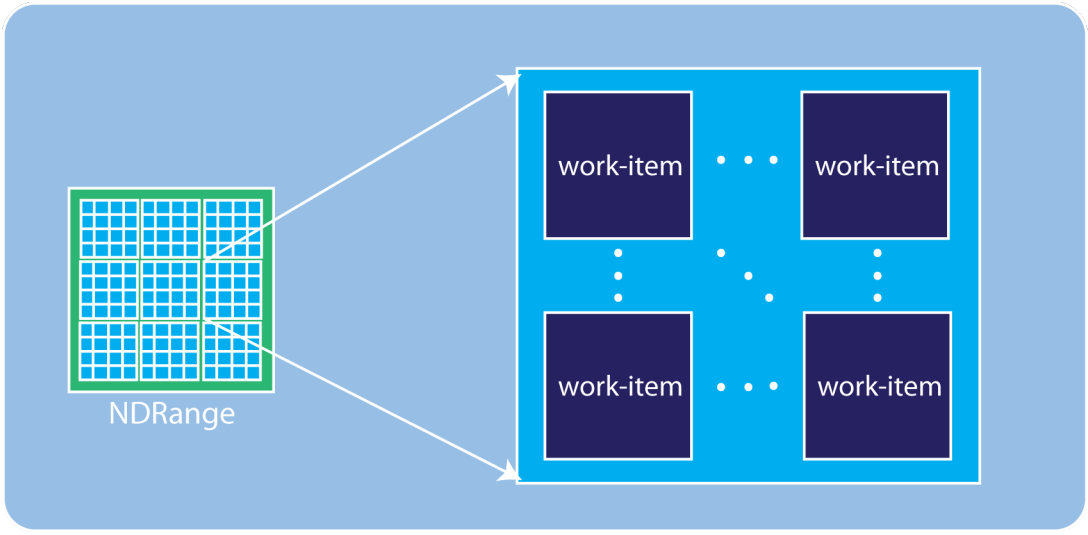

Figura 4.5: Modelo de ejecución de OpenCL

- **Objetos de memoria:** Un conjunto de objetos de memoria que pueden ser accedidos desde la máquina y los dispositivos OpenCL.

Este contexto se crea y manipula utilizando funciones de la API OpenCL por el *host*, el cual además crea una estructura de datos llamada **cola de comandos** que gestiona la ejecución de los *kernels* en los dispositivos. Tras la creación de la cola de comandos, el *host* introduce comandos en la cola que van a ser planificados en los dispositivos dentro del contexto. Los comandos se pueden dividir en tres grandes grupos:

- **Comandos de ejecución del *kernel*:** Ejecución de un *kernel* en los elementos de proceso de un dispositivo.
- **Comandos de memoria:** Transferencia de datos entre objetos de memoria.
- **Comandos de sincronización:** Gestiona el orden de ejecución de los comandos.

La ejecución de los comandos dentro del dispositivo se produce de manera asíncrona entre la máquina y el dispositivo. Existen dos modos de ejecución: en orden,

donde los comandos se lanzan en el orden que aparecen en la cola de comandos y se completan en ese mismo orden, y fuera de orden, donde los comandos se lanzan en orden, pero pueden no finalizan en ese orden. La ejecución de los comandos generan objetos del tipo evento. Estos eventos se utilizan para controlar la ejecución entre comandos y para coordinar la ejecución entre la máquina y los dispositivos. Es posible asociar varias colas a un mismo contexto, las cuales se ejecutarán de manera concurrente e independiente sin tener métodos de sincronización dentro de OpenCL.

En cuanto a la memoria utilizada por OpenCL, se pueden diferenciar cuatro tipos diferentes:

- **Global:** Permite lecturas y escrituras a todos los *work-items* de todos los *work-groups*.
- **Constante:** Se mantiene constante durante la ejecución de un *kernel*. La máquina es la encargada de reservar e inicializar los objetos en esta memoria.
- **Local:** Es una región local a un *work-group*. Se puede utilizar para reservar memoria que van a compartir todos los *work-items* de un *work-group*.
- **Privada:** Es una región privada de un *work-item* determinado.

La Figura 4.6 muestra la organización de la memoria en OpenCL y los diferentes accesos permitidos desde el dispositivo y desde la máquina que funciona como *host*.

La programación en OpenCL puede expresarse de dos maneras diferentes: explotanndo el paralelismo a nivel de datos o a nivel de tareas presentes en la aplicación. El paralelismo a nivel datos define una operación como una secuencia de instrucciones aplicadas a varios elementos de un objeto en memoria. El paralelismo a nivel tareas define un modelo en el cual una instancia de un kernel se ejecuta sin tener en cuenta ningún índice de espacios.

Por otro lado, debido al paralelismo y la concurrencia, surge la necesidad de la sincronización. En OpenCL hay dos dominios que necesitan sincronización: los *work-items* dentro de un mismo *work-group*, y los comandos apilados en las colas

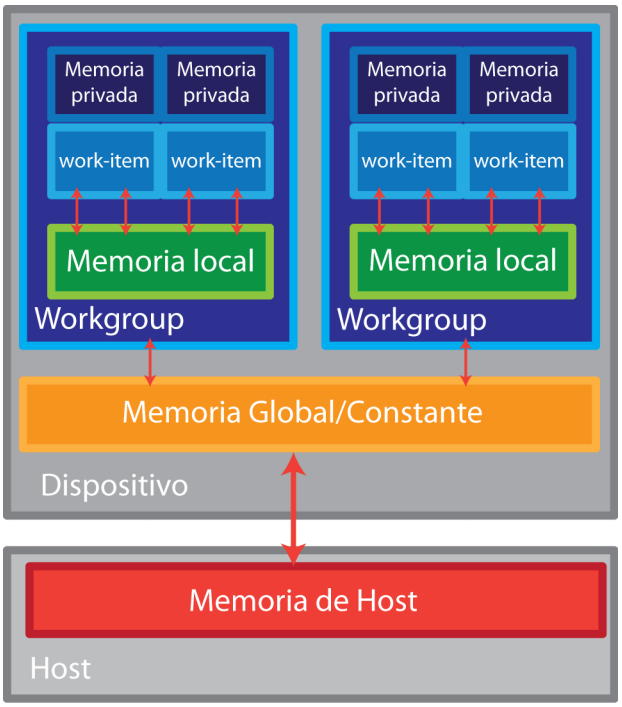

Figura 4.6: Modelo de memoria de OpenCL (Adapatado de [12])

de comandos de un contexto determinado. Para el primer caso, la solución más rápida es la utilización de una barrera para todo el *work-group*, la cual deben ejecutar todos los elementos del *work-group* antes de que ninguno de ellos continúe ejecutando más allá de dicha barrera. No existe la sincronización entre varios *work-groups*. Los puntos de sincronización entre colas de comandos son de dos tipos:

- **Barrera de la cola de comandos:** Esta barrera asegura que todos los comandos apilados antes de ella han sido ejecutados y los resultados son visibles para todos los comandos que se ejecuten tras la misma. Se utiliza para sincronizar comandos en una misma barrera.
- **Espera de eventos:** Todas las funciones de la API que apilan comandos devuelven un evento que lo identifica junto con la memoria que actualizan. Por ello, cada comando ejecutado que espere ese evento se asegura no continuar ejecutando hasta que termine la función asociada al evento.

Capítulo 5

Descripción Informática

5.1. Especificación de Requisitos

La funcionalidad de la API, así como todos los detalles de la misma queda recogidos en los siguientes requisitos.

5.1.1. Requisitos funcionales

Los requisitos referidos al funcionamiento de la API de cara al usuario son los siguientes:

1. La API generada debe recubrir la especificación de OpenCL 1.0
2. La API generada debe ser compatible con los principales sistemas operativos soportados por las diferentes SDK.
   a) La API generada debe ser compatible con Windows XP 32 bits.
   b) La API generada debe ser compatible con Ubuntu 9.10 32 bits.
   c) La API generada debe ser compatible con Ubuntu 9.10 64 bits.

3. La API generada debe ser compatible con las SDK de los principales fabricantes.

   a) La API generada debe ser compatible con la SDK de NVIDIA.

   b) La API generada debe ser compatible con la SDK de ATI.

4. La API generada debe mantener un rendimiento comparable al obtenido con la SDK original.

5. La API generada debe ser fiel, en la medida de los posible, al estándar original

6. La API generada debe seguir el paradigma de la programación orientada a objetos en la medida de lo posible.

7. Las interacciones entre OpenCL y Java deben ser transparentes al usuario.

8. La API generada debe controlar los posibles errores producidos en OpenCL.

5.1.2. Requisitos no funcionales

Los requisitos necesarios para el correcto funcionamiento de la API, pero que no forman parte de su funcionalidad son los siguientes:

1. La API generada debe soportar todos los dispositivos soportados por las SDK originales.

2. Los requisitos hardware deben ser los mismos que los de las SDK originales.

3. Es necesario disponer de una máquina virtual de Java para poder utilizar la API.

4. Es necesario disponer de la librería original de OpenCL para poder utilizar la API.

5. Para la representación de los resultados de manera gráfica, será necesario disponer de JOGL.

5.2. Diseño e Implementación

En este capítulo se pasa a describir todos los detalles referentes al diseño e implementación de la API. Como se ha mencionado, consistirá en un fichero JAR que permitirá realizar las llamadas a funciones OpenCL, que a su vez recupera dichas funciones de una librería dinámica. Por tanto, en este apartado se aborda la generación de la API desde la creación de la librería dinámica hasta la finalización de la implementación de la especificación completa, así como las decisiones tomadas.

5.2.1. Arquitectura de JavaOpenCL

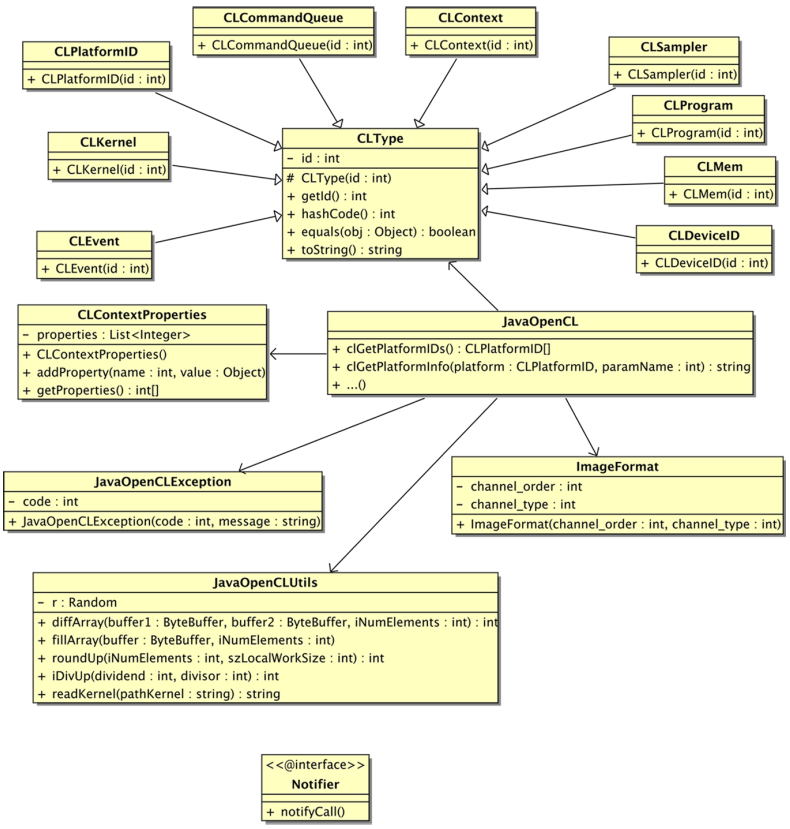

Figura 5.1: Diagrama UML del proyecto

La Figura 5.1 muestra el diagrama de clases del proyecto completo así como las relaciones entre las diferentes clases que lo forman. A continuación se presenta una breve descripción de cada una de las clases.

- **CLType:** Representa un tipo básico de OpenCL, como puede ser un *buffer* o una cola de comandos. Almacena el identificador proporcionado por OpenCL. Las subclases de `CLType` tienen la misma funcionalidad, pero diferenciando entre los tipos disponibles en OpenCL.
- **CLContextProperties:** Representa una lista de propiedades que pueden recibir los contextos al ser creados. Internamente contiene una lista de propiedades cuyo último elemento es 0 (definido por el estándar OpenCL).
- **JavaOpenCL:** Clase principal de la API. Contiene toda la especificación de OpenCL implementada en Java. Es la clase encargada de la comunicación con la librería original mediante JNI.
- **JavaOpenCLException:** Excepción de la API que se lanza en el momento en el que se produce un error asociado a OpenCL.
- **ImageFormat:** En OpenCL se trata de una estructura que almacena datos sobre el formato de una imagen que se va a procesar, por lo que en JavaOpenCL se ha transformado en una clase que almacena la misma información.
- **JavaOpenCL:** Conjunto de funciones muy comunes implementadas en las diferentes SDK portadas a fin de que sean compatibles con JavaOpenCL.
- **Notifier:** Interfaz que debe implementar un programador para utilizar un *callback* en JavaOpenCL.

5.2.2. Uso de JavaOpenCL

Para que el lector pueda comprender la necesidad de las diferentes fases del proyecto, a continuación se incluye un ejemplo de JavaOpenCL que muestra su uso.

Para ello se ha partido de un código base, cuya funcionalidad es rellenar un vector con los índices del mismo. El código fuente es el mostrado a continuación:

```
1  import static es.gavab.javapencl.JavaOpenCL.*;
2  import static es.gavab.javapencl.JavaOpenCLUtils.*;
3  import static es.gavab.javapencl.*;
4  import java.nio.ByteBuffer;
5  import java.nio.ByteOrder;
6
7  public class HelloWorld {
8     public static void main(String[] args) {
9        try {
10          final int arrayLength = 10;
11
12          CLContext context =
13             clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, null);
14          CLMem outCL = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
15                                null, (Integer.SIZE/8)*arrayLength);
16          CLDeviceID[] devices =
17             (CLDeviceID[]) clGetDeviceIDs(CL_PLATFORM_NVIDIA, CL_DEVICE_TYPE_ALL);
18          String[] source = new String[1];
19          source[0] = readKernel(".\\kernels\\HelloWorld.cl");
20          CLProgram program = clCreateProgramWithSource(context, source);
21          clBuildProgram(program, null, null, new NotifierBuild(){
22             public void notifyBuild() {
23                System.out.println("Este␣es␣el␣callback␣de␣la␣funcion");
24             }
25          });
26          CLKernel kernel = clCreateKernel(program, "hello");
27          clSetKernelArg(kernel, 0, 4, outCL);
28          CLCommandQueue cq = clCreateCommandQueue(context, devices[0], 0);
29          int[] global = new int[1];
30          int[] local = new int[1];
31          global[0] = (Integer.SIZE/8)*arrayLength;
32          local[0] = 1;
33          CLEvent event =
34             clEnqueueNDRangeKernel(cq, kernel, 1, null, global, local, null);
35          CLEvent[] lista = new CLEvent[1];
36          lista[0] = event;
37          ByteBuffer lectura = ByteBuffer.allocateDirect
38                ((Integer.SIZE)/8*arrayLength).order/ByteOrder.nativeOrder());
39          clEnqueueReadBuffer(cq, outCL, true, 0,
40                (Integer.SIZE)/8*arrayLength, lectura, lista);
41          clReleaseKernel(kernel);
42          clReleaseProgram(program);
43          clReleaseCommandQueue(cq);
44          clReleaseContext(context);
45          lectura.clear();
46          for (int i=0;i<arrayLength;i++) {
47             System.out.println("␣"+lectura.getInt());
```

```
48            }
49        } catch (JavaOpenCLException e) {
50            System.out.println("EXCEPCION");
51            System.out.println("Codigo: "+e.getCode());
52            e.printStackTrace();
53        }
54    }
55 }
```

El primer paso para poder desarrollar una aplicación JavaOpenCL es importar la librería al proyecto actual, de manera que sea posible acceder a la API. Tras esto, es necesario importar las clases de JavaOpenCL que van a ser utilizadas. Para mayor comodidad, se importan todos los métodos estáticos de las clases JavaOpenCL y JavaOpenCLUtils, de manera que el acceso a los mismos sea directo, en lugar de seguir el formato `Clase.nombreMetodo()`.

En la línea 10 se declara una variable `arrayLength` que se utilizará para definir la longitud del vector. Tras esto, es necesario crear el contexto, con la sentencia:

```
1    CLContext context =
2        clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, null);
```

Con ello se indica que el programa va a ser ejecutado en un dispositivo GPU. Después es necesario declarar el *buffer* donde se almacenarán los resultados de la ejecución:

```
1    CLMem outCL = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
2                                 null, (Integer.SIZE/8)*arrayLength);
```

La función `clCreateBuffer()` recibe como argumento el tamaño en *bytes* del *buffer* que será creado. El objetivo es que el código se pueda ejecutar en todas las versiones de JavaOpenCL, además de que el programador no tiene por qué saber el tamaño de un entero en bytes en Java. Por eso se utiliza la expresión `Integer.SIZE/8`, que devolverá el tamaño que ocupa un entero en bytes en Java.

Tras esto, se ejecuta una sentencia que consulta qué dispositivos GPU se encuentran disponibles para la aplicación. Debido a que se utiliza la SDK de NVIDIA, ajustamos la búsqueda a dispositivos NVIDIA:

```
1  CLDeviceID[] devices =
2     (CLDeviceID[]) clGetDeviceIDs(CL_PLATFORM_NVIDIA, CL_DEVICE_TYPE_ALL);
```

Los identificadores de los dispositivos disponibles quedarán almacenados en la variable devices. Tras la obtención de los dispositivos, se crea el programa que será ejecutado, junto con el *kernel*. Se supone que el fichero con el *kernel* se encuentra en una carpeta llamada *kernels* dentro del directorio de la aplicación.

```
1  String[] source = new String[1];
2  source[0] = readKernel(".\\kernels\\HelloWorld.cl");
3  CLProgram program = clCreateProgramWithSource(context, source);
```

Es importante destacar que en la variable `source` estarán almacenados todos los *kernels* que se necesiten cargar (en este caso sólamente uno). Tras la obtención del código fuente del *kernel*, se pasa a la creación del programa, desde ése código fuente. El siguiente paso es compilar el programa, con la instrucción:

```
1  clBuildProgram(program, null, null, new NotifierBuild(){
2     public void notifyBuild() {
3        System.out.println("Este␣es␣el␣callback␣de␣la␣funcion");
4     }
5  });
```

Esta función, además de compilar el programa, ejecuta la función `notifyBuild` durante su propia ejecución, a modo de *callback*. Es posible insertar `null` en la posición de este parámetro si no se necesita dicha función. Para poder definir el *callback* en Java, se utiliza una clase anónima que debe implementar una interfaz que contiene el método `notifyBuild`.

El siguiente paso es crear el *kernel* y establecer sus argumentos, en este caso un único argumento, el vector a rellenar:

```
1  CLKernel kernel = clCreateKernel(program, "hello");
2  clSetKernelArg(kernel, 0, (Integer.SIZE/8), outCL);
```

El parámetro `hello` hace referencia al nombre de la función del *kernel* que se va a ejecutar en el dispositivo. El argumento que se va a utilizar es una referencia al

vector de salida, por lo que su tamaño es el tamaño de una referencia, el equivalente a un entero.

Es necesario entonces crear la cola de ejecución donde se encolarán los comandos a ejecutar, en el dispositivo seleccionado:

```
1   CLCommandQueue cq = clCreateCommandQueue(context, devices[0], 0);
```

Esta cola de comandos recibirá el contexto sobre el que va a trabajar, así como el dispositivo donde se van a ejecutar dichos comandos. El siguiente paso es ordenar la ejecución del *kernel* en la cola de comandos:

```
1   int[] global = new int[1];
2   int[] local = new int[1];
3   global[0] = (Integer.SIZE/8)*arrayLength;
4   local[0] = 1;
5   CLEvent event =
6      clEnqueueNDRangeKernel(cq, kernel, 1, null, global, local, null, true);
7   CLEvent[] lista = new CLEvent[1];
8   lista[0] = event;
```

Es importante remarcar varios aspectos. En primer lugar, las variables `global` y `local` hacen referencia al número de *work-items* y de *work-groups* que se van a utilizar. En este caso, al ser un ejemplo sencillo bastará con un sólo *work-group* que contenga un número de *work-items* igual al tamaño del *buffer* a procesar. Tras ello, se ordena la ejecución del *kernel*, y de esa llamada se recupera un evento que se va a utilizar para sincronizar la ejecución.

Tras la ejecución del *kernel*, es necesario leer los resultados obtenidos, y liberar los recursos reservados en el dispositivo gráfico.

```
1   clEnqueueReadBuffer(cq, outCL, true, 0,
2         (Integer.SIZE)/8*arrayLength, lectura, lista, false);
3   clReleaseKernel(kernel);
4   clReleaseProgram(program);
5   clReleaseCommandQueue(cq);
6   clReleaseContext(context);
```

La sentencia `clEnqueueReadBuffer` recupera el *buffer* de la memoria del dispo-

sitivo gráfico para poder leerla desde memoria principal. El penúltimo parámetro de dicha llamada es la lista de eventos creada anteriormente, por lo que esta función no va a ejecutarse hasta que finalicen todas las sentencias acumuladas en la lista.

Cabe destacar el uso de un bloque `try-catch` que envuelve el código para comprobar cualquier posible error que se genere en alguno de los comandos anteriores.

Por último, aunque no resulte significativo para el desarrollo de la API, se incluye el código del *kernel* para una mejor comprensión del ejemplo:

```
1  __kernel void hello(__global int * out)
2  {
3     size_t tid = get_global_id(0);
4     out[tid] = tid+1;
5  }
```

5.2.3. Generación de librería dinámica

El primer paso para generar la API es disponer de una librería dinámica con la cual se puedan realizar las llamadas a las funciones del estándar de OpenCL. Para ello es necesario crear un proyecto en C que genere la librería dinámica, y un proyecto Java que contenga la declaración de los métodos a incluir en la librería. En este caso se ha utilizado EclipseGavab 2.0, por su integración con ambos lenguajes, utilizando el compilador GCC en Ubuntu y MinGW en Windows, ambos software libre. La configuración de los proyectos está incluida en el Anexo 1 (7.1).

Para generar dicha librería es necesario conocer cuáles van a ser las funciones que se van a incluir en la misma. Para ello, se declara el método que va a estar incluido en la librería con el modificador `native`. Esto hace que el intérprete de Java detecte que esa función va a estar declarada dentro de una librería, por lo que no hay que implementarla. Tras esto, hay que generar el fichero de cabeceras que contiene la declaración de esa librería, pero en código C. Esto es posible realizarlo de manera sencilla utilizando la herramienta `javah` proporcionada con el JDK. El comando para generar este fichero es el siguiente:

```
javah -jni -d Directorio/Proyecto/C ClaseJava
```

Cabe destacar que `Directorio/Proyecto/C` será el directorio del proyecto C que va a generar la librería, y `ClaseJava` será la clase Java compilada que contiene el método nativo. De esta manera se generará en el directorio del proyecto C un fichero `.h` que contendrá la cabecera de la función. El siguiente paso es implementar dicha función.

Para ello se crea un fichero `.c` donde es necesario copiar la cabecera generada. Para la implementación de esta función no se va a utilizar JNI, por lo que no es necesario conocer el significado de los parámetros `JNIEnv` ni `jclass` aún. Tras tener la función implementada, se pasa a la compilación del proyecto C, que en caso de ser correcta, genera la librería dinámica (fichero `.dll` para Windows, `.so` para Ubuntu). Para que el proyecto Java pueda cargar esta librería, existen dos opciones. La primera de ellas es copiarla a un directorio incluido en el *path* de las librerías del sistema operativo, como puede ser `WINDOWS\SYSTEM32` en Windows. La otra forma, que es la que se corresponde con la utilizada en este proyecto, es la inclusión de la librería en el directorio del proyecto Java.

Con la librería disponible en el proyecto, es posible utilizar desde una clase Java las funciones nativas implementadas. Para ello, se carga la librería desde Java mediante el siguiente comando:

```
1 static {
2    System.loadLibrary("libreria");
3 }
```

Este bloque es necesario situarlo al comienzo de la clase Java que va a invocar los métodos nativos, de manera que cada vez que se ejecute la aplicación Java se cargue la librería. En el código, `libreria` debe ser sustituido por el nombre de la librería generada. A partir de este punto la aplicación Java está preparada para invocar los métodos nativos. Para ello, sólo es necesario invocarlos como si de métodos normales de Java se tratara, ya que será el compilador el encargado de obtener ése método de la librería.

5.2.4. Generación de una API básica

Con la librería generada, el siguiente paso es crear una API que haga uso de la misma. Con esta finalidad se crea una clase Java similar a la anterior, en la que se encuentre incluida la carga de la librería, así como un recubrimiento de todos los métodos presentes en la misma. Para este recubrimiento sólo es necesario implementar un método que llame al método nativo, que también debe estar declarado en la clase Java. Por último, se necesita exportar este proyecto Java en forma de fichero JAR, que contendrá a la API anterior.

Una vez generado el fichero JAR, se incluye en un nuevo proyecto Java, el cual utilizará la API. Además, es necesario importar los métodos de la librería, de manera que sean visibles en el nuevo proyecto. Esto se puede realizar mediante el comando `import` de Java, sin ninguna configuración adicional. Para una mayor comodidad a la hora de programar, se han declarado los métodos que recubren a los nativos como estáticos, de manera que no es necesaria la creación de ningún objeto para invocarlos, además de simplificar su uso, importando dichos métodos de la siguiente manera:

```
1 import static es.gavab.javaopencl.JavaOpenCL.*;
```

Así, para invocar a los métodos ya no será necesario utilizar el nombre del paquete (`JavaOpenCL.metodo()`), sino que bastará con el nombre del método (`metodo()`). Así, se dispone de la estructura básica del proyecto, una API que el usuario debe importar y que se encarga de la gestión de información entre Java y C.

5.2.5. Uso de JNI

Para el uso de JNI es necesario tener en cuenta diversos aspectos, ya que la sintaxis del código C va a cambiar ligeramente. Estos cambios se van a situar principalmente en las cabeceras de las funciones y en las llamadas a funciones JNI.

Aunque las cabeceras las produce **javah** automáticamente, es importante en-

tender el significado de las mismas para poder implementar el código. Las cabeceras de las funciones nativas siguen el siguiente esquema:

```
1 JNIEXPORT tipo_retorno JNICALL Java_NombreClase_NombreMetodo
2  (JNIEnv * env, jobject object, ...);
```

Siempre van precedidas de `JNIEXPORT tipo_retorno JNICALL`, donde `tipo_retorno` es el tipo de datos que va a recibir Java al invocar el método nativo, por ejemplo `jint`. `NombreClase` y `NombreMetodo` son los nombres de la clase Java y del método nativo respectivamente, tal como aparecen en el código Java. La parte más importante corresponde a los parámetros que recibe siempre cualquier función nativa. Estos parámetros son:

- **JNIEnv * env:** Se trata de un puntero a una tabla que almacena referencias a todas las funciones proporcionadas por JNI. Éstas son todas las funciones que se necesitan para interactuar con la máquina virtual y trabajar con objetos y métodos Java. Cabe destacar que en todas las funciones se debe pasar como parámetro el propio `env`. Por ejemplo, `(*env)->GetStringUTFChars(env, javaString, 0);` convierte un *String* de Java en una cadena de caracteres de C. La Figura 5.2 muestra la representación de un elemento de este tipo:

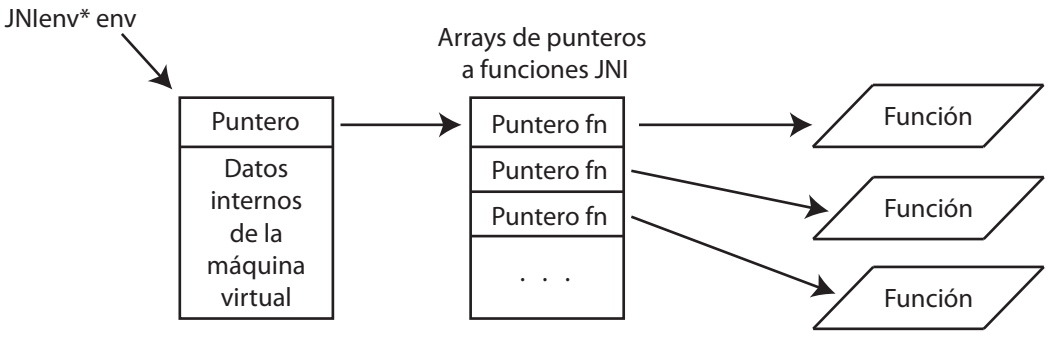

Figura 5.2: Estructura de los elementos `JNIEnv`

- **jobject object:** Este argumento tiene dos significados. Si se trata de un método de instancia, actúa como un puntero `this` al objeto Java. Si es un método de clase, se trata de una referencia `jclass` a un objeto que representa la clase en la que están definidos los métodos estáticos. En el caso de JavaOpenCL no se va a utilizar, debido a que la clase que va a invocar los métodos va a ser siempre la misma, y no tiene que participar en el código nativo.

Los cambios referentes a las llamadas en JNI se deben principalmente a que hay que tener en cuenta que los datos recibidos vienen de Java, no desde otra función en C. Por ello, hay que realizar diferentes conversiones, como por ejemplo la de un `String` de Java a una cadena de caracteres en C, y estas conversiones se realizan a través de las funciones que presenta JNI.

5.2.6. Comunicación con OpenCL

El siguiente paso es la comunicación de Java con OpenCL. Para ello, es muy importante saber como se va a gestionar el intercambio de datos entre OpenCL y Java, ya que en OpenCL se dispone de tipos, como los punteros, que no tienen correspondencia directa con Java. La Figura 5.1 muestra la comparativa de los principales cambios en ambos lenguajes:

Punteros

La gestión de los punteros desde Java es una decisión de diseño imprescindible para la interacción con OpenCL. Una primera aproximación ha sido la utilización de los tipos básicos de Java para representarlos. Para los sistemas de 32 bits, se ha utilizado el tipo `int` de Java para representarlos, y el tipo `long` para los de 64 bits. Esta decisión viene dada por el tamaño de los punteros en el lenguaje C en cada uno de los tipos de sistemas operativos, que se corresponden con los tipos elegidos en Java. Tras comprobar el correcto funcionamiento del uso de estos tipos, es conveniente ofrecer una capa de abstracción que haga la utilización de

Tipo en OpenCL / C	**Tipo en Java**
Punteros	`int` (32 bits) ó `long` (64 bits)
Buffers de datos	`ByteBuffer`
Callbacks	Clase `Notifier`
Errores	Excepciones
CL_TRUE, CL_FALSE	booleanos
SDK de utilidades OpenCL	SDK de utilidades JavaOpenCL
Uso de OpenGL para representar gráficos	Uso de JOGL para representar gráficos

Cuadro 5.1: Tabla correspondencia de tipos OpenCL - Java

estos tipos transparente al usuario.

Aprovechando las ventajas de Java, se ha considerado la creación de una clase Java por cada uno de los tipos de OpenCL que representan un puntero, las cuales heredarán de una superclase `CLType` que encapsula el elemento común a todos los tipos, que será el valor del puntero. Además, la clase `CLType` implementa el método `equals` para comparar si dos elementos son iguales a través de su identificador, así como el método `hashCode` para que el código *hash* de cada objeto de esta clase se genere a partir del identificador.

El uso de una clase de este tipo debe ser simple para el programador. Con este motivo, la librería dinámica obtiene el valor del identificador a partir de la llamada correspondiente a OpenCL. Tras obtener dicho identificador, será la API Java la encargada de encapsular el valor obtenido dentro de un objeto de JavaOpenCL. Todas las clases incluyen un constructor para crearlas que reciben como parámetro el identificador. El usuario no va a ser el que utilice estos constructores, ya que será la clase que gestiona la comunicación con OpenCL la encargada de obtener el valor del puntero, crear el objeto pertinente y devolver la referencia al objeto creado. De esta forma, el usuario tan sólo deberá declarar un objeto del tipo necesario y asignarle el resultado de la llamada JavaOpenCL, sin preocuparse de la reserva de memoria del mismo.

Buffers de datos

En OpenCL, los buffers de datos utilizados generalmente se crean e inicializan desde el código C, no desde el kernel. Por eso es necesario presentar la funcionalidad de la creación de estos *buffers* de datos de la manera más similar a la realizada en OpenCL. Una primera aproximación puede consistir en utilizar *arrays* de Java, la manera más sencilla. En este caso los *arrays* no son una buena solución debido a su bajo rendimiento, ya que para poder utilizarlos desde JNI sería necesario realizar una copia del *array*. Esta copia, teniendo en cuenta los tamaños de los *arrays* que se suelen utilizar en OpenCL, reduce el rendimiento e incluso imposibilita la ejecución en algunos casos. Utilizando *arrays* la copia es necesaria ya que son reservados en la memoria de la Máquina Virtual de Java, y para poder acceder a ellos desde C se necesita que estén almacenados en memoria nativa.

Por este motivo en este proyecto se propone el uso de `ByteBuffers`, presentes en el paquete `java.nio`. La utilidad de estos *buffers* reside principalmente en la reserva de memoria que nos permiten. Hay dos tipos de `ByteBuffer`, los directos (creados con el método `allocateDirect()`), o los no directos (creados con el método `allocate()`). La creación de los dos tipos siempre necesita como argumento el tamaño del *buffer* en bytes, pero con una diferencia. Cuando se utiliza un *buffer* directo, la máquina virtual ejecutará las operaciones nativas de entrada salida sobre ese buffer directamente sobre él, mientras que con los no directos, se realizará una copia del mismo. Por esto, en JavaOpenCL se deben utilizar *buffers* directos para un mejor rendimiento.

Otra de las ventajas de los *buffers* directos es que la memoria se reserva directamente en memoria nativa, lo que facilita su acceso desde JNI. Este es el motivo por el cual se utilizarán `ByteBuffers` para gestionar *buffers* de datos en JavaOpenCL, que desde el código C serán gestionados a través de las funciones que presenta JNI para este propósito (`GetDirectBufferAddress()`, `GetDirectBufferCapacity()`, etc.). Estas funciones permiten traducir el `ByteBuffer` a un *array* de tipos básicos en C.

También resulta interesante su método `slice()`, que puede ser de gran utilidad en caso de necesitar realizar aritmética de punteros desde Java. Este método crea

un *buffer* con el contenido del *buffer* que lo invoca. Además, tiene la particularidad de que los cambios realizados sobre este nuevo *buffer* se verán reflejados en el original. Aunque no recibe argumentos, la creación se realiza a partir de la posición indicada por el atributo `position` del *buffer* original. Por ello, para acceder a una posición concreta basta con modificar el atributo `position` con el desplazamiento buscado e invocar al método `slice()`.

Callbacks

En OpenCL hay varias funciones que incluyen *callbacks*, esto es, punteros a funciones que se van a ejecutar al llamar a las funciones que los incluyen. En un principio esto puede parecer imposible de realizar desde Java, debido a la ausencia de punteros, y mucho menos de punteros a funciones. Además, debe ser un método de alguna manera intuitivo para los usuarios de Java, de manera que no sea necesaria la gestión de punteros o similares. Por esto, la opción elegida es la expuesta a continuación.

Primero se ha creado una interfaz Java que representa una clase que va a contener el *callback*. Esta interfaz incluye un método `notifyCall` que es el que se va a ejecutar mediante el `callback`, de manera que es este método el que debe implementar el usuario. En el código nativo, se ha implementado una función que recibe como parámetro el objeto que implementa la interfaz anterior. Esta función, utilizando la API proporcionada por JNI, realiza las siguientes operaciones:

1. Extrae el identificador de la clase a la que pertenece el objeto, para conocer el tipo de función. Esta extracción se realiza mediante la función `FindClass()`, que recibe como parámetro la signatura de la clase buscada. La signatura de una clase o un método es una cadena de caracteres que identifica unívocamente a esa clase o método.

2. Extrae el identificador del método que va a ser invocado. Esta extracción se realiza mediante la función `GetMethodID()`, que recibe como parámetro el nombre del método y su signatura de tipos.

3. Invoca el método extraído anteriormente, utilizando la función `CallVoidMethod()`, que recibe como parámetro al objeto que contiene el método a invocar y el método que se va a invocar.

4. Elimina la referencia creada, ya que no va a volver a ser utilizada. Esto es importante, ya que de otra manera esta referencia no podría ser eliminada, quedando alojada en memoria. En aplicaciones con varios *callbacks*, esto puede influir en el rendimiento de la aplicación.

Ahora es necesario especificar como va a utilizar el usuario estos `callbacks`. La forma más sencilla es con el uso de clases anónimas. Para esto, lo único que debe hacer el usuario es, a la hora de introducir un callback, declarar en ese mismo instante la clase a la que va a pertenecer (implementando la interfaz correspondiente), e implementar el método `notify` incluido en dicha clase. De esta manera se instancia en el mismo momento de la invocación el objeto que va a contener la información del `callback` que debe ejecutarse.

Por último, es necesario permitir el uso de parámetros de la aplicación dentro de esos callbacks. Para conseguir dicha funcionalidad, hay que declarar previamente constantes que almacenen el valor de dichos parámetros. Es decir, si queremos utilizar como parámetro el identificador del programa, declaramos la constante:

```
final int programID = program.getId();
```

En el fragmento anterior se puede suponer que `program` es un objeto de la clase `CLProgram` creado anteriormente. De esta manera, dentro del método `notifyCall` podremos utilizar esta constante para obtener la información deseada.

Dentro de la colección de ejemplos existe una clase llamada *EjemploError* que contiene una de estas funciones. La función es la que se muestra en el siguiente fragmento de código:

```
1       clBuildProgram(program, null, null,
2          new Notifier() {
3             @Override
4             public void notifyCall() {
```

```
5                 System.out.println("Program␣=␣"+progConst);
6             }
7         })
```

En ese ejemplo se muestra como se debe incluir un *callback* en cualquier función de JavaOpenCL que lo acepte. De esta forma, al ejecutar la función *clBuildProgram* se mostrará por pantalla el mensaje indicado en el *callback*. Esto puede ser útil en caso de necesitar información a la que sólo se tiene acceso desde dentro de la llamada a la función.

Tratamiento de excepciones

En OpenCL, para poder comprobar si una función se ha ejecutado correctamente o no, es necesario comparar el valor devuelto por las funciones con la constante `CL_SUCCESS` que indica que la ejecución ha sido correcta. Esta comparación, si se porta directamente a Java, puede generar un código engorroso además de que en Java los errores no deben comprobarse con el tipo de retorno de la función, para eso existen las excepciones.

Para aprovechar la potencia de Java, lo natural es utilizar las excepciones que presenta el lenguaje, de manera que para el usuario, cualquier método JavaOpenCL no ejecutado correctamente generará una excepción, y será el usuario el que decida si desea o no tratarla. En caso de ignorarla, se detendrá la ejecución del programa, ya que la ejecución incorrecta de un sólo comando OpenCL lleva a la obtención de resultados erróneos en la gran mayoría de los casos.

El lanzamiento de dichas excepciones se produce en el momento en que se detecta el error, esto es, en el código nativo. Para ello ha sido necesaria la implementación de una función que lance la excepción con un código de error y un mensaje determinado. Además, es necesario crear una clase Java encargada del lanzamiento de la excepción, la clase `JavaOpenCLException`. Esta clase hereda de `RuntimeException` y tiene como atributo el código de error. La función que gestiona la excepción desde C es `throwException`, y su funcionalidad es la siguiente:

- Obtiene una referencia a la clase `JavaOpenCLException` con la función `FindClass()`, sólo si es la primera vez que se lanza una excepción, en caso contrario ya tiene la referencia.

- Obtiene una referencia al constructor de la excepción, con la función `GetMethodID()`, sólo si es la primera vez, en caso contrario ya tiene la referencia.

- Crea un objeto de tipo `JavaOpenCLException` utilizando la función `AllocObject()`.

- Invoca al constructor de la clase `JavaOpenCLException` con la función `CallNonvirtualVoidMethod()`.

- Declara una variable de tipo `jthrowable`, que representa las excepciones desde JNI, y le asigna el objeto creado anteriormente.

- Lanza la excepción con la función `Throw()`.

Como se puede observar, no se eliminan las referencias creadas. Esto es principalmente para que en el caso de que el usuario maneje las excepciones no sea necesario obtener una referencia cada vez, ya que va a ser siempre la misma. De esta manera se mejora el rendimiento de la gestión de excepciones.

Así, al producirse una excepción en JavaOpenCL, el usuario obtendrá un código de error, así como la descripción de la especificación original del motivo de la excepción, para poder comprobarlo en la especificación, en caso de no conocer el motivo de la misma.

Booleanos

En OpenCL para simular los valores booleanos se utilizan dos constantes, `CL_TRUE` y `CL_FALSE`, que simulan los valores *verdadero* y *falso*. Para portar esto a Java, la elección ha sido utilizar los valores booleanos que proporciona Java, `true`

y `false`, y que sea la API la encargada de traducir dichos valores a los proporcionados por OpenCL, mediante dos funciones diferentes, una para transformar de booleano de OpenCL a booleano de Java y otra para el sentido contrario.

Lectura de kernels

La especificación de OpenCL no proporciona ninguna funcionalidad para la lectura de los kernels, por lo que es el usuario el encargado de abrir los ficheros correspondientes, leer el contenido y cargarlo al programa OpenCL. En las diferentes SDK, esto ha sido solucionado mediante el uso de funciones que proporciona la propia SDK para cargar estos kernels.

Para poder realizar algo similar en Java, se ha implementado una función `readKernel()` que recibe como argumento la ruta (absoluta o relativa) donde se encuentra el fichero del kernel que se desea leer. De esta manera, para cargar el código de un kernel, tan sólo es necesario utilizar dicha función, que lo almacenará en un `String` a través del valor de retorno de la misma. Esta función se encuentra dentro de la clase `JavaOpenCLUtils`.

Representación de gráficos

Para representar gráficos en OpenCL se utiliza OpenGL, aunque aún no está disponible la interoperabilidad directa (acceder desde OpenCL a los datos de OpenGL y viceversa). Por eso, se utiliza la copia interna de los datos, de manera que no perjudique al rendimiento. Esta copia implica que un mismo *buffer* reservado en memoria principal puede ser utilizado tanto en OpenCL como en OpenGL. En Java existe JOGL, una API de OpenGL para Java, que proporciona la misma funcionalidad que OpenGL. Debido a que la relación entre JavaOpenCL y JOGL es similar a la existente entre OpenCL y OpenGL se ha decidido utilizar JOGL como API para representar gráficos desde JavaOpenCL.

5.2.7. Gestión de eventos

En OpenCL se pueden considerar dos tipos de sincronización: la sincronización entre los diferentes *work-items* de un mismo *work-group* y la sincronización entre diferentes comandos encolados en colas de comandos de un mismo contexto. Para la sincronización entre *work-items* se utiliza sincronización por barrera, de manera que ningún *work-item* continúa ejecutando tras la barrera hasta que el resto hayan llegado a ella. Por lo tanto, para este caso no es necesario añadir nada especial a JavaOpenCL.

Sin embargo, para la sincronización de comandos existen dos formas de sincronización: por barrera, que funciona igual que el caso anterior, y por eventos. La sincronización por eventos se basa en que cada comando encolado puede retornar un evento, de manera que el usuario puede hacer que los siguientes comandos esperen la finalización del evento para continuar ejecutando. En OpenCL esto se realiza de manera que si el usuario quiere obtener ese evento para sincronizar, debe pasar la dirección de memoria de un elemento *cl_ event*, y en caso contrario, debe pasar **NULL**.

En OpenCL estas funciones necesitaban que el evento se recibiera como un parámetro ya que el valor de retorno indicaba que la función se ha ejecutado correctamente o no. Ya que en Java este problema está solucionado con el uso de excepciones, lo más sencillo es utilizar el valor de retorno de la función en JavaOpenCL para devolver el evento, y que sea el usuario el que decida si utilizarlo o no, dependiendo de si es necesario sincronizar diferentes comandos.

Esta primera aproximación a priori parece totalmente válida, pero tiene un grave problema. Utilizando este método, aunque el usuario no vaya a utilizar los eventos, siempre van a ser creados. En OpenCL la creación de los eventos implica que el usuario se va a encargar de controlarlos, lo que incluye su liberación, mediante la función `clReleaseEvent`. En JavaOpenCL el usuario puede ignorar este evento, lo que en principio no causa ningún problema, debido a la escasa cantidad de memoria utilizada por un objeto `CLEvent`, la cual puede ser gestionada por el recolector de Java.

Pero existe un problema más grave, ya que en OpenCL si se crea un evento pero no se libera, todos los elementos que puedan estar asociados a ese evento, como pueden ser *buffers*, que suelen tener un tamaño considerable, quedan sin liberar, aunque el usuario explícitamente elimine sus referencias mediante su función `clReleaseMemObject`. Es decir, de esta manera un *buffer* asociado a un evento liberar que se libere explícitamente, quedará inútil para el usuario pero ocupando memoria en el dispositivo en el que se haya creado. Por lo tanto, aparece la necesidad de obligar al usuario a liberar los eventos en caso de que se creen.

Por ello la solución elegida es la de utilizar un parámetro de tipo *boolean* donde el usuario puede indicar si quiere crear (`true`) o no (`false`) el evento asociado. En caso de querer crear dicho evento, debe hacerse resposable de su liberación, y en caso de no crear el evento, debe ignorarlo, ya que el tipo de retorno será un valor que no tiene ninguna información. Por tanto, todas las funciones que puedan generar eventos, recibirán un último parámetro booleano donde el programador indicará si quiere o no gestionar el evento generado.

Así se consigue una solución cercana a Java, al aprovechar el tipo de retorno, a la vez que la gestión de la memoria queda gestionada de la manera más transparente al usuario.

5.2.8. SDK de utilidades JavaOpenCL

Debido a que todas las SDK disponibles para OpenCL proporcionan una librería que contiene las funciones más utilizadas, se ha desarrollado una biblioteca similar adaptada a JavaOpenCL, para que sea lo más similar posible a la incluida en la SDK original. En este conjunto de funciones se incluyen, entre otras, la inicialización de un *ByteBuffer* con valores aleatorios, así como la comparación de los resultados obtenidos en dos *ByteBuffer* diferentes.

Esta librería está implementada en la clase *JavaOpenCLUtils* que contiene todos sus métodos estáticos, al igual que JavaOpenCL, para que no se necesaria la instanciación de ningún objeto para utilizarlo. La lista completa de las funciones que se pueden encontrar es la que se muestra a continuación:

- `int diffArray(ByteBuffer buffer1, ByteBuffer buffer2, int iNumElements)`: Función utilizada para comparar dos `ByteBuffers` tras realizar operaciones sobre ellos en *OpenCL* y *Java*.

- `int diffArray(ByteBuffer buffer1, ByteBuffer buffer2, int iNumElements, float range)`: Tiene la misma funcionalidad que la anterior, sólo que en este caso no se comparan que sean exactamente iguales, si no que sean similares dentro de un rango.

- `void fillArray(ByteBuffer buffer, int iNumElements)`: Inicializa un `ByteBuffer` con números aleatorios

- `int roundUp(iNumElements, int szLocalWorkSize)`: Obtiene el múltiplo de `szLocalWorkSize` inmediatamente superior a `iNumElements`. Es útil cuando el número de elementos a tratar no coincide con el `szLocalWorkSize` elegido.

- `int iDivUp(int dividend, int divisor)`: Obtiene la división entera de dos elementos, redondeada al entero mayor.

- `String readKernel(String pathKernel)`: Lee un fichero que contiene el código de un *kernel* de OpenCL y lo convierte a *String* para utilizarlo para construir el programa.

5.3. Ejecución de JavaOpenCL

La ejecución de una función en JavaOpenCL puede dividirse en tres etapas principales:

1. **Llamada a una función por parte del usuario:** Esta etapa abarca desde que el usuario realiza una llamada a una función de JavaOpenCL hasta que esta llamada es transformada por la API. La función de esta etapa es la de transformar todos los tipos y clases propios de Java a tipos más cercanos al lenguaje C, para obtener una mayor eficiencia.

2. **Llamada a la función por parte de la API:** Tras realizar dicha transformación, se realiza una llamada a la función nativa implementada en la **.dll** con los argumentos ya transformados en un formato válido para la librería.

3. **Llamada a la función OpenCL por parte de la librería:** Por último, en la librería se realiza una última conversión de los datos entrantes a tipos primitivos de C y de OpenCL, para finalmente realizar la llamada original de OpenCL.

El retorno al usuario se realiza de la forma inversa.

1. **Resultados OpenCL:** Se obtienen los resultados proporcionados por la llamada a OpenCL y se transforman en tipos primitivos de Java dentro de la librería.

2. **Conversión de datos:** Tras obtener los datos en forma de tipos primitivos de Java en la API, se crean los objetos necesarios dependiendo de la llamada, transformando esos datos en los tipos de JavaOpenCL.

3. **Resultados finales:** El usuario recoge los datos en el formato de JavaOpenCL tras realizar la llamada correspondiente.

Un posible ejemplo de ejecución que implica la creación de un buffer en JavaOpenCL es la que aparece en la Figura 5.3. Como se puede ver en dicha figura, las dos primeras etapas, encargadas de transformar la función desde Java a C, se ejecutan sobre la máquina virtual, y una vez los datos están preparados para pasar a memoria nativa, se ejecuta la última etapa en el *host*.

5.4. Problemas encontrados

A lo largo del proyecto han surgido diversos problemas que finalmente han sido solucionados. Los siguientes problemas representan aquellos que más trabajo y tiempo han requerido para ser solucionados, debido a su complejidad:

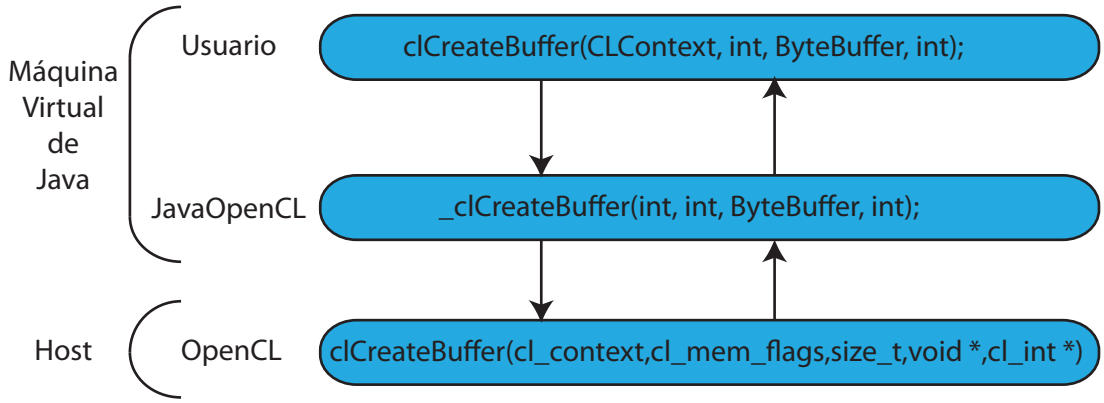

Figura 5.3: Ejemplo de ejecución de la función clCreateBuffer

Compatibilidad con la SDK de AMD

Al comienzo del proyecto AMD no disponía de una SDK para programar OpenCL bajo estos dispositivos. Con la salida de la versión beta de dicha SDK en Diciembre de 2009, se realizaron las pruebas correspondientes, obteniendo resultados erróneos en todos los ejemplos. El problema residía en que AMD liberó su SDK dando soporte tan sólo al compilador de Microsoft Visual Studio para Windows, con un gran número de *bugs* conocidos pero no solucionados. Por este motivo no fue posible la portabilidad del proyecto a la SDK de AMD.

Mediante la utilización de los foros de desarrolladores, se solicitó a la compañía soporte para el compilador GCC, muy utilizado en la actualidad por un gran número de desarrolladores, lo que quedó reflejado mediante el apoyo a dicha propuesta por parte de otros desarrolladores.

En Enero de 2009 se liberó la versión final de la SDK, pero con otro problema: el driver necesario para utilizar la SDK no era totalmente compatible con todos los dispositivos, además de ser una versión beta. Aunque el soporte a GCC seguía

sin estar disponible, se realizaron las mismas pruebas que la versión anterior, obteniendo exactamente los mismos resultados, por lo que no ha sido posible portar el proyecto para el uso de la SDK de ATI hasta el momento.

Gestión de la memoria

La experimentación realizada puede dividirse en dos pruebas diferentes. En la primera de ellas se realiza la experimentación ejecutando cada uno de los ejemplos seleccionados una sola vez, mientras que en la segunda se ejecuta cada ejemplo 20 veces, para comprobar la influencia de la carga de trabajo en JavaOpenCL.

Al realizar la segunda prueba apareció un problema con la gestión de memoria que Java realiza con los ByteBuffer, que hacía que Java lanzara una excepción del tipo *OutOfMemoryException* cuando en realidad debería llamar al recolector de basura, encargado de liberar la memoria antes de lanzar una excepción de este tipo.

Este problema aparece reflejado en la base de datos de *bugs* de Java (*bug id*=4857305), en el que se indica que eventualmente es posible que al reservar una gran cantidad de memoria en un *ByteBuffer* que sobrepase la memoria disponible, se lance la excepción previamente indicada, terminando la ejecución de la aplicación.

Para solucionar este problema de una manera transparente al usuario, se ha optado por incluir una llamada al recolector de basura en cada método *clRelease* de la API, ya que en el momento de liberar un buffer de OpenCL, no va a ser utilizado más veces, por lo que puede ser liberado también de la memoria principal.

Capítulo 6

Resultados Experimentales

La experimentación realizada con la API JavaOpenCL comienza por la adaptación de una colección de ejemplos de la SDK de NVIDIA a JavaOpenCL, para realizar una comparativa frente a OpenCL. Se ha seleccionado un conjunto de la SDK original debido a que estos ejemplos ya se encuentran en un estado estable y optimizado. Debido a esto, sólo es necesario portar los ejemplos literalmente, sabiendo que éstos se encuentran ya optimizados.

Además, la SDK de NVIDIA proporciona ejemplos con diferentes finalidades, desde la comprobación del funcionamiento hasta la demostración de las ventajas y desventajas de OpenCL, pasando por algunos ejemplos gráficos que demuestran su sencilla comunicación con OpenGL (JOGL en el caso de JavaOpenCL).

Debido a la gran cantidad de ejemplos disponibles en la SDK y a sus continuas actualizaciones, se ha elegido un conjunto que incluya código con diferentes objetivos. Entre estos objetivos se encuentran la comprobación de un correcto funcionamiento de OpenCL/JavaOpenCL, el cálculo de la mejora obtenida al utilizar OpenCL/JavaOpenCL frente a C/Java, o la representación de gráficos cuyo origen proviene de ciertos cálculos realizados en OpenCL/JavaOpenCL.

La colección de ejemplos portados se enumera a continuación:

- **BitonicSort:** Implementación del algoritmo *Bitonic Mergesort*. Este algoritmo se utiliza principalmente para crear una red de ordenación, un modelo matemático abstracto de cables y módulos de comparación que se utiliza para ordenar una secuencia de números.

- **Black-Scholes:** El modelo Black-Scholes se emplea para estimar el valor de una opción europea para la compra o venta de acciones en una fecha futura. Este ejemplo realiza dichos cálculos sobre un conjunto de opciones europeas.

- **DCT8x8:** Implementación de la Transformada de coseno discreta. Expresa una secuencia finita de varios puntos como resultado de la suma de distintas señales sinusoidales. Se suele utilizar para la compresión de datos, como por ejemplo compresión de vídeo (MPEG-4) o imágenes (JPEG)

- **DeviceQuery:** Es el ejemplo más sencillo. Tan sólo consulta información del dispositivo gráfico a través de JavaOpenCL.

- **DotProduct:** Realiza el producto escalar a un conjunto de pares de vectores de entrada.

- **MatrixMul:** Realiza la multiplicación de matrices. Este ejemplo es muy significativo, ya que está orientado hacia una configuración *multi-gpu*, es decir, un dispositivo gráfico que incluye dos GPU en lugar de uno sólo para aumentar la potencia del dispositivo.

- **MatrixTranspose:** Calcula de manera eficiente la traspuesta de una matriz.

- **MatVecMul:** Realiza la multiplicación de un vector y una matriz.

- **Scan:** Dado un array de números, calcula un nuevo array donde cada elemento es la suma de todos los elementos anteriores del vector original.

- **SimpleGLJOGL:** Este ejemplo demuestra la compatibilidad de JavaOpenCL con JOGL, calculando en JavaOpenCL una gráfica 3D de la curva del seno y representándola mediante JOGL.

- **SimpleMandelbrot:** Mediante JavaOpenCL se crea un fractal de Mandelbrot, que posteriormente es renderizado por JOGL.

Para medir el rendimiento de los distintos ejemplos se han realizado dos tipos de pruebas:

- La primera basada en una únicas ejecución de cada ejemplo.

- Debido a que los tiempos obtenidos en la primera prueba no son significativos (menores de 1 segundo en todos los casos), se ha tomado la decisión de realizar una segunda prueba que ejecuta 20 veces cada ejemplo, para medir cómo afecta la continua ejecución de código a JavaOpenCL frente a OpenCL.

Además, cada tipo de prueba se ha ejecutado dos veces. En una de ellas se han incluido las reservas de memoria principal y de vídeo y en la otra tan sólo el fragmento de código correspondiente a trabajo sobre la GPU (memoria de vídeo). Gracias a estas ejecuciones es posible comprobar como puede influir el trabajo en CPU por parte de Java frente al trabajo de CPU por parte de C, incluyendo las reservas de memoria e inicialización de los datos en ambos lenguajes.

Es importante destacar que el objetivo de estas pruebas es comprobar el rendimiento de JavaOpenCL frente a OpenCL, y comprobar la posible pérdida de rendimiento de JavaOpenCL frente a OpenCL al igual que el de Java frente a C. Es por este motivo por lo que en los resultados en varios casos se puede comprobar que los resultados en C son más rápidos. Esto es debido a que los ejemplos tienen como finalidad comprobar el correcto funcionamiento de JavaOpenCL y OpenCL, por lo que habrá algunos de ellos que no conllevan una cantidad de cómputo suficiente para superar a C o incluso a Java.

6.1. Resultados numéricos

En la tabla 6.1 pueden verse los resultados obtenidos cuando se incluyen los tiempos de reserva e inicialización de memoria, tanto de vídeo como principal.

	JavaOpenCL		OpenCL		Java		C	
	1 it.	20 it.	1 it.	20 it.	1 it.	20 it.	1 it.	20 it.
Black Scholes	1062	20344	828	17077	2078	41437	1140	22999
DCT8x8	609	11547	515	9687	719	10577	140	2796
DotProduct	2047	39077	1187	23546	2813	50235	921	17859
MatrixTranspose	1828	36234	1812	35640	78	1469	31	531
MatVecMul	484	9094	484	9046	500	46	15	31
VectorAdd	1500	29652	843	16484	1656	32109	437	8750

Cuadro 6.1: Tabla comparativa incluyendo las reservas de memoria

En la tabla 6.2 aparecen reflejados los resultados que se han obtenido al ejecutar los ejemplos portados excluyendo de los mismos la reserva e inicialización de los datos en memoria principal.

	JavaOpenCL		OpenCL		Java		C	
	1 it.	20 it.	1 it.	20 it.	1 it.	20 it.	1 it.	20 it.
Black Scholes	500	8406	484	9578	1469	29249	781	15719
DCT8x8	453	8047	453	7813	281	5500	47	781
DotProduct	390	7015	485	9031	906	17687	47	812
MatrixTranspose	1843	36140	1797	35156	31	500	16	140
MatVecMul	468	8953	453	9062	0	0	0	0
VectorAdd	453	8515	453	8204	563	10203	31	640

Cuadro 6.2: Tabla comparativa sin incluir las reservas de memoria

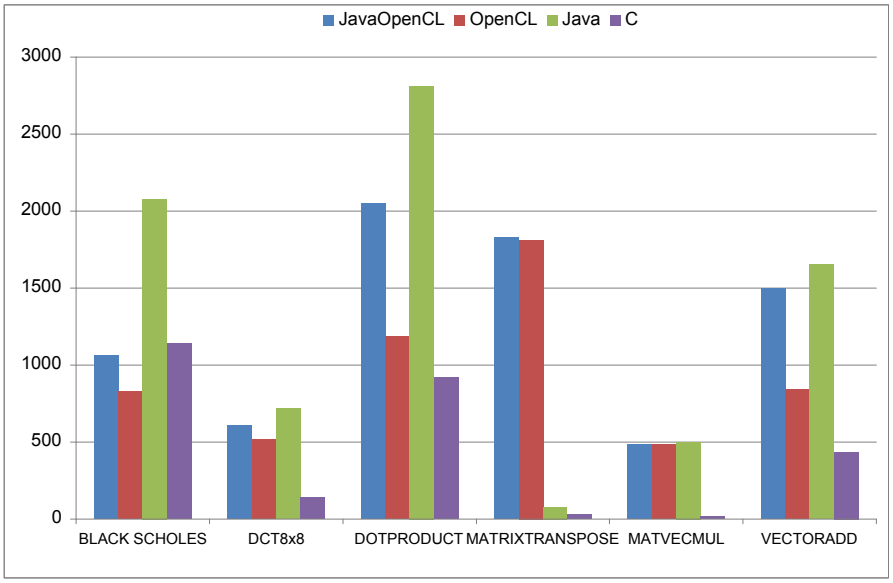

Figura 6.1: Ejecución de una iteración con gestión de memoria

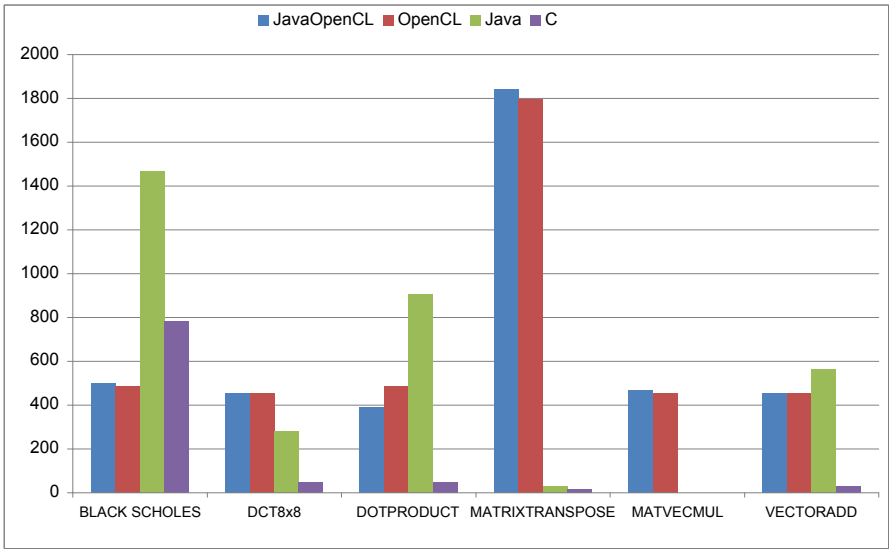

Figura 6.2: Ejecución de una iteración sin gestión de memoria

En un primer análisis, se puede comprobar que la API de JavaOpenCL ha cumplido el objetivo principal, el cual se basa en obtener un rendimiento similar

al de OpenCl, de manera que el usuario no sea capaz de detectar una gran pérdida de rendimiento al utilizar JavaOpenCL.

Tras este análisis inicial, es necesario comprobar los resultados más directos, que son aquellos que va a percibir el usuario, en los que cada ejemplo se ejecuta una vez por completo, esto es, incluyendo las reservas previas de memoria y su inicialización, realizada con Java y C, respectivamente. Para ello es necesario referirse a la gráfica 6.1. En estos resultados, se puede comprobar que por regla general, el tiempo empleado por JavaOpenCL es mayor que el empleado por OpenCL. Sin embargo, es posible que esto sea debido a la reserva e inicialización de memoria en Java (la cual no ha sido optimizada de ninguna manera, para mantenerse fiel al ejemplo original en C), que sea más ineficiente que la reserva e inicialización en C.

Para poder comprobar esto, es necesario analizar la gráfica correspondiente a los resultados de la ejecución de una sola iteración de cada ejemplo, de manera que se puedan comparar las pérdidas de rendimiento debidas a la ejecución en CPU. Al analizar dicha gráfica (6.2), se puede confirmar que la hipótesis es correcta, ya que en la misma el tiempo de ejecución de JavaOpenCL es muy similar al de OpenCL en todos los casos, sin grandes variaciones.

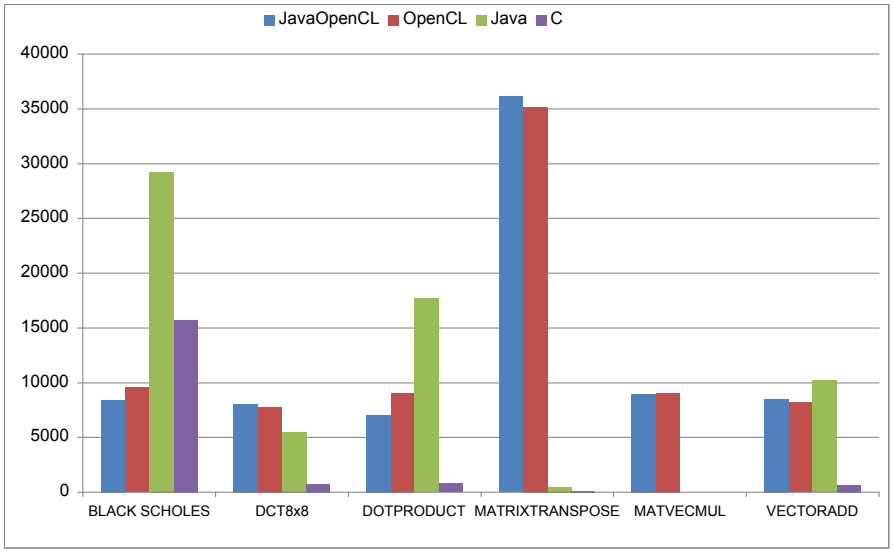

Figura 6.3: Ejecución de 20 iteraciones sin gestión de memoria

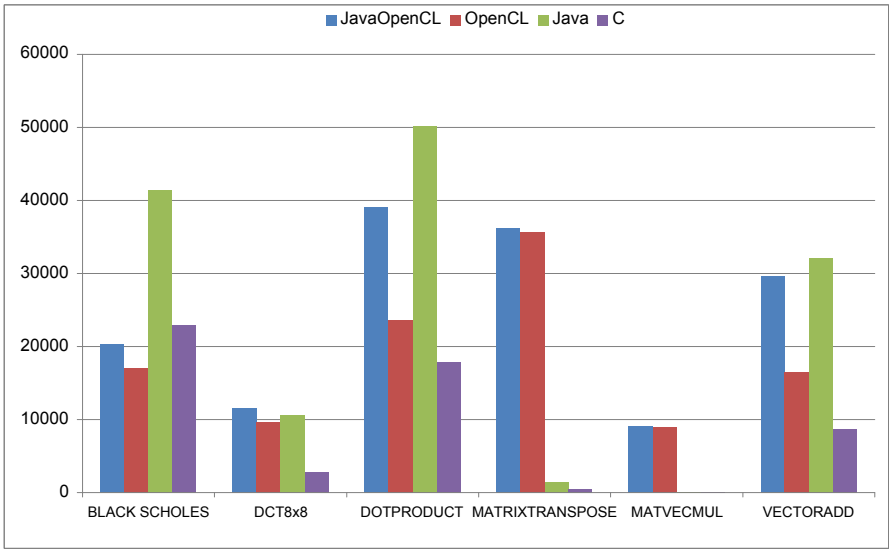

Figura 6.4: Ejecución de 20 iteraciones con gestión de memoria

Es posible también comprobar que una continua ejecución no afecta al rendimiento de JavaOpenCL frente a OpenCL. Esto se puede comprobar en las gráficas 6.3 y 6.4. En dichas gráficas se puede comprobar que los resultados se mantienen respecto a los obtenidos en la primera prueba.

Resulta interesante también mostrar las ventajas y desventajas de la utilización de JavaOpenCL en lugar de Java. Si se tiene en cuenta la motivación del uso de OpenCL, se puede suponer que será recomendable utilizar JavaOpenCL en lugar de Java en los casos en los que se vayan a realizar cálculos intensivos y paralelizables, y se utilizará Java en los casos de pequeños volúmenes de datos que vayan a realizar operaciones ligeras. Además, si tan sólo se tiene en cuenta el cálculo de los resultados, ignorando la previa reserva de la memoria en el *host*, los resultados son aún más favorables hacia JavaOpenCL. Esto se puede comprobar en las gráficas 6.1, 6.2, 6.3 y 6.4.

Como se puede comprobar, ejemplos como *BlackScholes* o *DotProduct*, los cuáles implican una gran cantidad de datos, obtienen resultados en JavaOpenCL que mejoran en gran proporción los obtenidos con Java. Sin embargo, si se analizan los resultados obtenidos por JavaOpenCL en ejemplos como MatrixTranspose(el

más notable) o MatVecMul, los cuáles no implican cálculo sobre una cantidad considerable de datos de entrada, se comprueba que el rendimiento es mayor utilizando Java. Los mismos resultados se pueden obtener al comparar los resultados de OpenCL frente a los de C.

El siguiente paso es comparar los resultados obtenidos utilizando varios dispositivos gráficos con diferentes prestaciones. Para esta experimentación se han utilizado los siguientes dispositivos:

- **NVIDIA GeForce GTX260:** 896 MB de memoria DDR3
- **NVIDIA GeForce 9800:** 2 x 512 MB de memoria DDR3
- **NVIDIA GeForce 8600 GTS:** 256 MB de memoria DDR3
- **NVIDIA GeForce FX 5600:** 256 MB de memoria DDR
- **NVIDIA GeForce 8800 GTX:** 768 MB de memoria DDR3

Para realizar esta comparación se ha utilizado el ejemplo que más cantidad de datos procesa (*BlackScholes*), para comprobar qué rendimiento ofrece cada dispositivo en casos extremos.

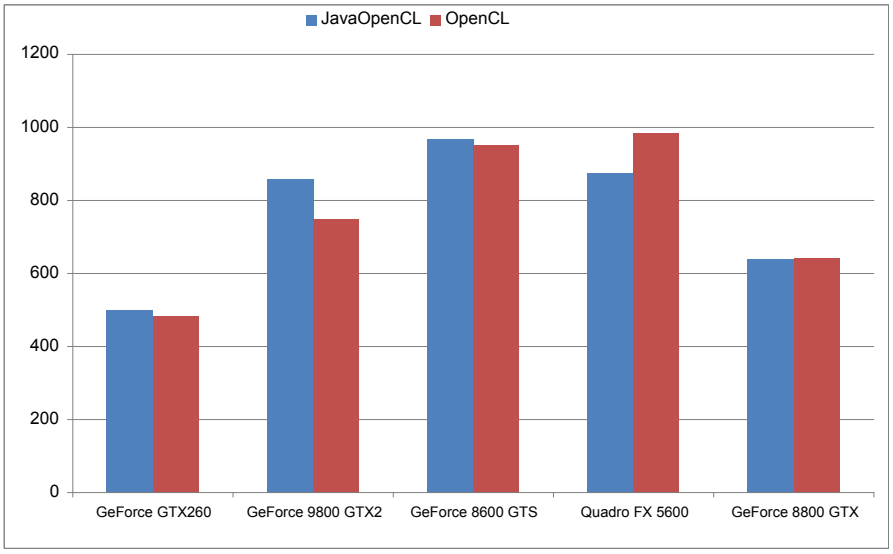

Figura 6.5: Ejecución de 1 iteración sin gestión de memoria

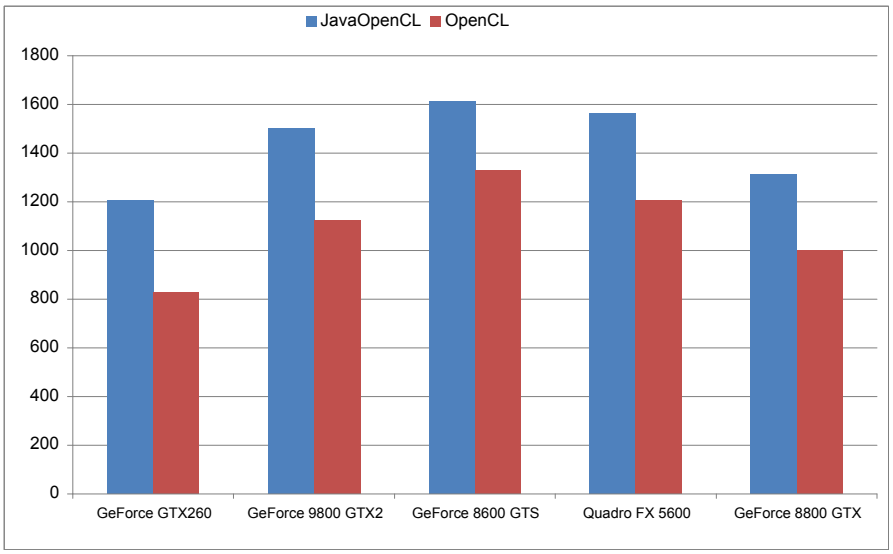

Figura 6.6: Ejecución de 1 iteración con gestión de memoria

Las gráficas 6.5 y 6.6 muestran los resultados obtenidos tras realizar la primera prueba, que implica una sola ejecución de cada ejemplo. En estos resultados es

posible comprobar que la gestión de la memoria en Java afecta a todos los dispositivos por igual. También se puede comprobar que JavaOpenCL no ofrece pérdida de rendimiento frente a OpenCL en la gráfica 6.5, que no incluye la gestión de memoria por Java.

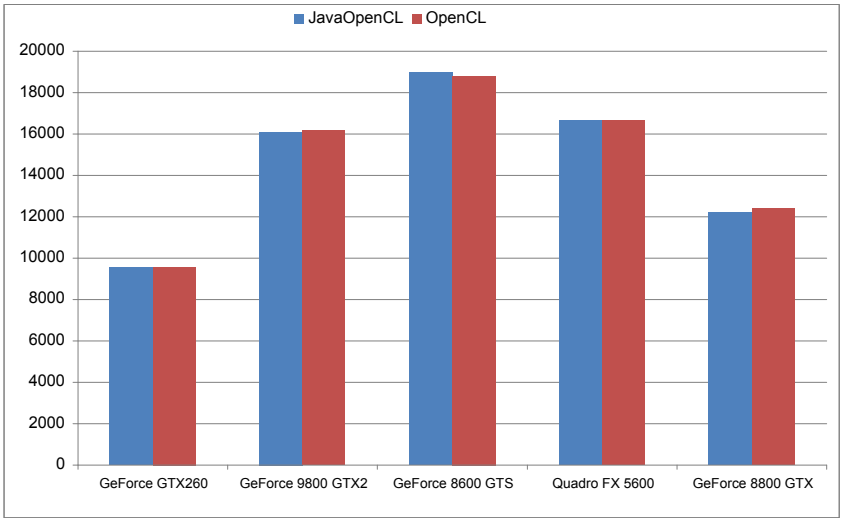

Figura 6.7: Ejecución de 20 iteraciones sin gestión de memoria

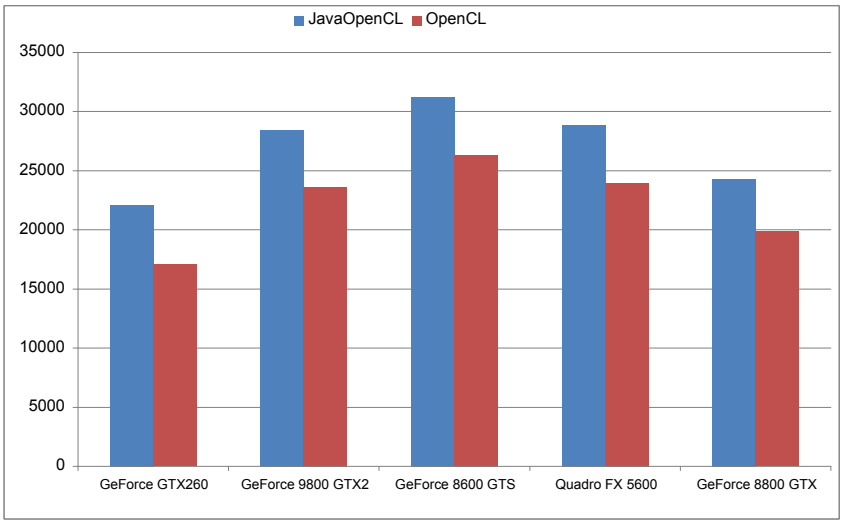

Figura 6.8: Ejecución de 20 iteraciones con gestión de memoria

La segunda prueba (20 iteraciones) realizada sobre los diferentes dispositivos (ver gráficas 6.7 y 6.8) proporciona unos resultados que confirman la eficiencia de JavaOpenCL frente a OpenCL, además de la influencia de la gestión de memoria por parte de Java (gráfica 6.8).

Por último, queda comparar el ejemplo *multi-gpu* para JavaOpenCL, que es aquél en el que se tiene un código que divide el trabajo entre los dispositivos. En este caso, se ha probado un ejemplo de multiplicación matricial ejecutándose sobre una tarjeta gráfica GeForce 9800 GX2, que cuenta con dos núcleos de procesamiento, frente al código correspondiente ejecutado sobre un único núcleo de un procesador E8400 Intel Core 2 Duo.

Si se analizan las gráficas 6.9 y 6.10 se puede comprobar que JavaOpenCL obtiene un rendimiento 25 veces mayor que Java, sin importar si se tiene en cuenta la gestión de memoria en Java o no. Es necesario comprobar que estos resultados se mantienen en la segunda prueba, con 20 iteraciones.

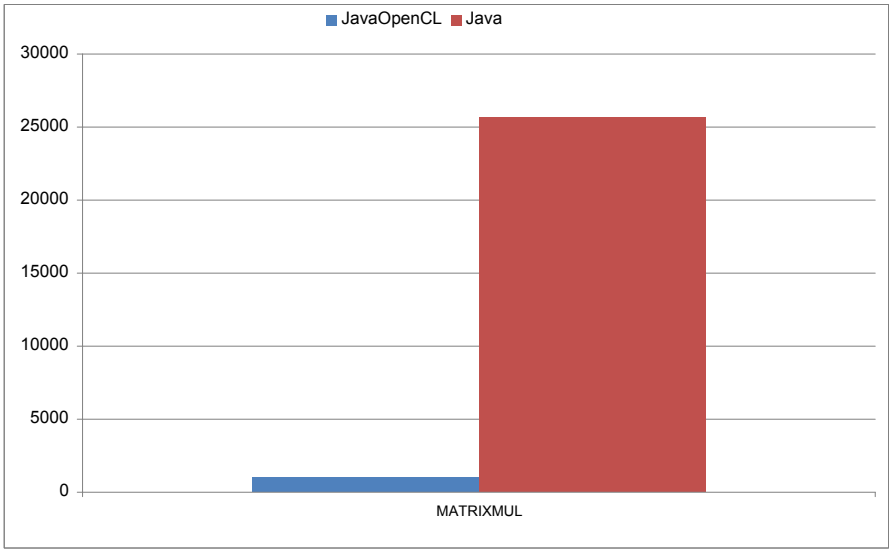

Figura 6.9: Ejecución de 1 iteración sin gestión de memoria

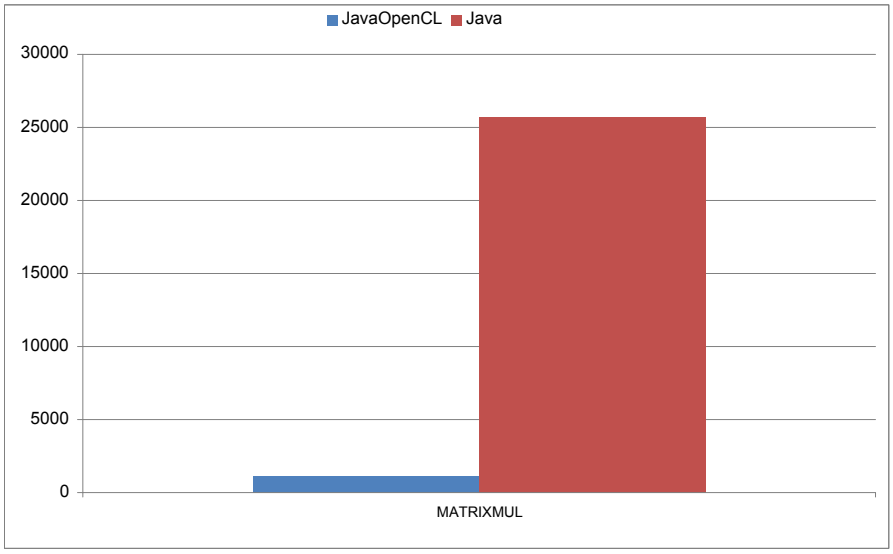

Figura 6.10: Ejecución de 1 iteración con gestión de memoria

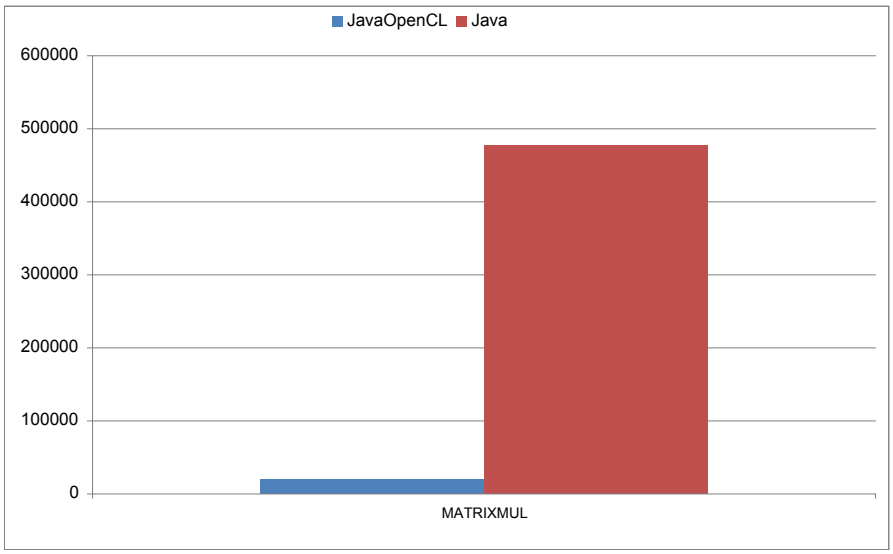

Figura 6.11: Ejecución de 20 iteraciones sin gestión de memoria

A la vista de los resultados obtenidos en la segunda prueba (ver gráficas 6.11 y 6.12) el *speedup* de JavaOpenCL frente a Java se mantiene en la misma proporción,

lo que hace que JavaOpenCL se una elección óptima para problemas con una gran cantidad de datos, que sean altamente paralelizables.

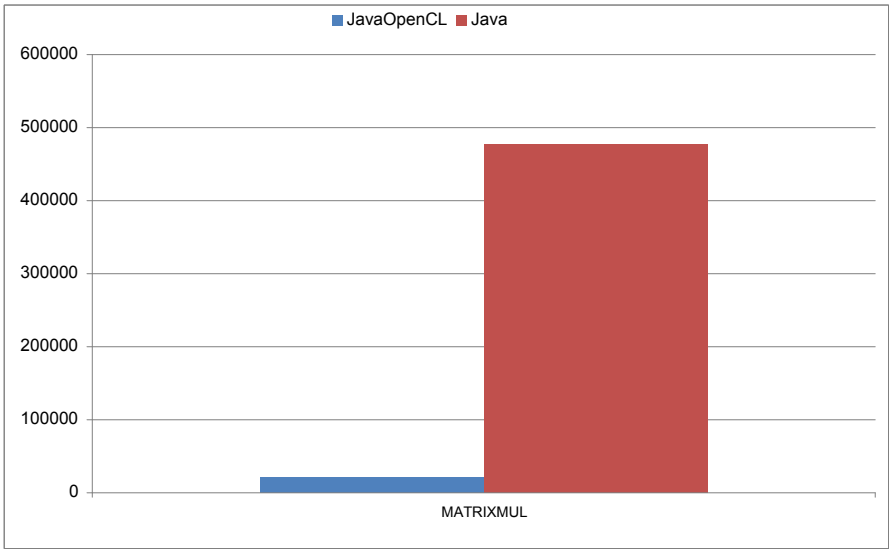

Figura 6.12: Ejecución de 20 iteraciones con gestión de memoria

Otro aspecto que se puede tener en cuenta en las comparativas es la pérdida de rendimiento con el paso de OpenCL a JavaOpenCL frente al paso de C a Java. Como se puede comparar en las gráficas anteriores la pérdida de rendimiento entre JavaOpenCL y OpenCL es prácticamente inexistente, por lo que portar una aplicación de OpenCL a JavaOpenCL no debe suponer ningún tipo de problema. Sin embargo, si se comprueba la pérdida obtenida al portar uno de los ejemplos de C a Java, se llega a la conclusión de que las pérdidas son considerables.

Capítulo 7

Conclusiones

Teniendo en cuenta los objetivos presentados en el Capítulo 6, se puede concluir que la API de JavaOpenCL cumple el requisito principal, basado en la generación de una API para Java que permita la programación sobre OpenCL. Gracias a la API de JavaOpenCL es posible desarrollar aplicaciones Java que se ejecuten sobre dispositivos gráficos con un rendimiento similar al obtenido con la plataforma original.

Además, esta API se ha hecho compatible con las principales plataformas utilizadas en la actualidad (Windows XP 32 bits, Ubuntu 9.10 32 y 64 bits), por lo que se consigue una cierta independencia del sistema operativo.

El segundo objetivo principal indica que la API debe ser fiel a la original, para evitar necesitar un tiempo prolongado de aprendizaje. Este objetivo ha sido cumplido, simplificando todo lo posible la API aprovechando las ventajas proporcionadas por Java. Esto se puede comprobar, por ejemplo, en que no es necesario indicar a una función JavaOpenCL el tamaño de los parámetros, ya que es la propia API la encargada de calcularlos, lo que simplifica en gran medida algunas de las funciones de OpenCL.

También se han realizado una serie de experimentaciones que prueban el correcto funcionamiento de la API JavaOpenCL, las cuales cumplían una doble función.

Por un lado, se utilizan para comprobar el correcto funcionamiento, tanto en rendimiento como en resultados, y además son muy útiles para familiarizarse con el entorno, ya que abarcan toda la API, de manera que siempre se pueden tener en cuenta a la hora de consultar referencias durante el desarrollo.

Durante el desarrollo del conjunto de pruebas, surgió la necesidad de disponer de un motor gráfico que sirviera para representar los resultados obtenidos por JavaOpenCL. Además, esto debía realizarse también en Java, de manera que el usuario no necesitara cambiar de plataforma para ello. Esta necesidad fue cubierta con la utilización de JOGL, el *binding* de OpenGL para Java.

Por último, cabe destacar la ausencia en el momento del desarrollo de una SDK completa por parte de AMD, de forma que no ha sido posible probar su correcto funcionamiento. Sin embargo, las decisiones de diseño de la API hacen que en el momento de disponer de una SDK estable de AMD, la compatibilidad con JavaOpenCL será inmediata. Esto es debido a que JavaOpenCL ha sido desarrollado basándose en el estándar, en lugar de en SDK proporcionada por una compañía concreta, por lo que futuras SDK liberadas por diferentes compañías deben ser soportadas por JavaOpenCL sin ningún problema, siempre y cuando cumplan con el estándar liberado por Khronos Group.

7.1. Trabajos futuros

Tras la obtención de una API que recubre todos los aspectos de la especificación original de OpenCL, se proponen los siguientes trabajos para su mejora y actualización:

- **Modelado Orientado a Objetos:** Una vez se dispone de una API que ofrece la misma funcionalidad que el original, se podrá proporcionar un modelo orientado a objetos de JavaOpenCL que la haga más cercana al paradigma utilizado por Java. Este modelado se llevaría a cabo organizando los métodos de la API actual en clases de manera que al final se disponga de dos API diferentes: una destinada a la programación imperativa, muy similar al

OpenCL original, y otra destinada a la programación orientada a objetos. De esta manera, un desarrollador Java sin conocimientos previos de OpenCL quizá esté más interesado en el modelado orientado a objetos, mientras que un desarrollador C acostumbrado al uso de OpenCL estará más interesado en la API imperativa.

- **Interoperabilidad OpenGL:** En la actual versión de OpenCL no se encuentra activa la interoperabilidad con OpenGL que permite que ambas herramientas puedan compartir memoria directamente, de manera que se eliminan transferencias entre memoria principal y memoria de vídeo. Por eso resulta interesante que cuando esta funcionalidad se encuentre disponible, se recubran las nuevas funciones para que JavaOpenCL también permita dicha interoperabilidad con APIs como JOGL.

- **Compatibilidad con AMD Stream SDK:** Debido a que en el momento de terminar el proyecto no había disponible una versión estable de la SDK propuesta por AMD no ha sido posible la comprobación del correcto funcionamiento de JavaOpenCL con la misma, por lo que en futuro será necesario comprobar, y adaptar en caso de ser necesario, la API de JavaOpenCL, aumentando así su portabilidad en cuanto al número de dispositivos soportados.

Anexos

Anexo 1: Configuración de Eclipse para desarrollar JavaOpenCL

En este anexo se expone la configuración de *EclipseGavab 2.0* para continuar el desarrollo de *JavaOpenCL*. Esta configuración no es necesario realizarla para utilizar *JavaOpenCL*, pero es imprescindible para continuar su desarrollo en trabajos futuros o para corregir *bug* que puedan surgir.

Añadir directorios a *GNU C*

Lo primero que se necesita es añadir los directorios de los ficheros de inclusión del JDK al compilador. Para ello, se accede a las propiedades del proyecto *C/C++*, apartado *C/C++ General*, y en el subapartado *Paths and Symbols*, en la pestaña *Includes*, se añaden los directorios correspondientes a los ficheros de cabeceras necesarios de JNI. Estos directorios son `/ruta_hasta_JDK/include` y `/ruta_hasta_JDK/include/win32` (deben aparecer en este orden en la lista de *Include directories*). La Figura 7.1 muestra como debe quedar la pestaña *Includes*.

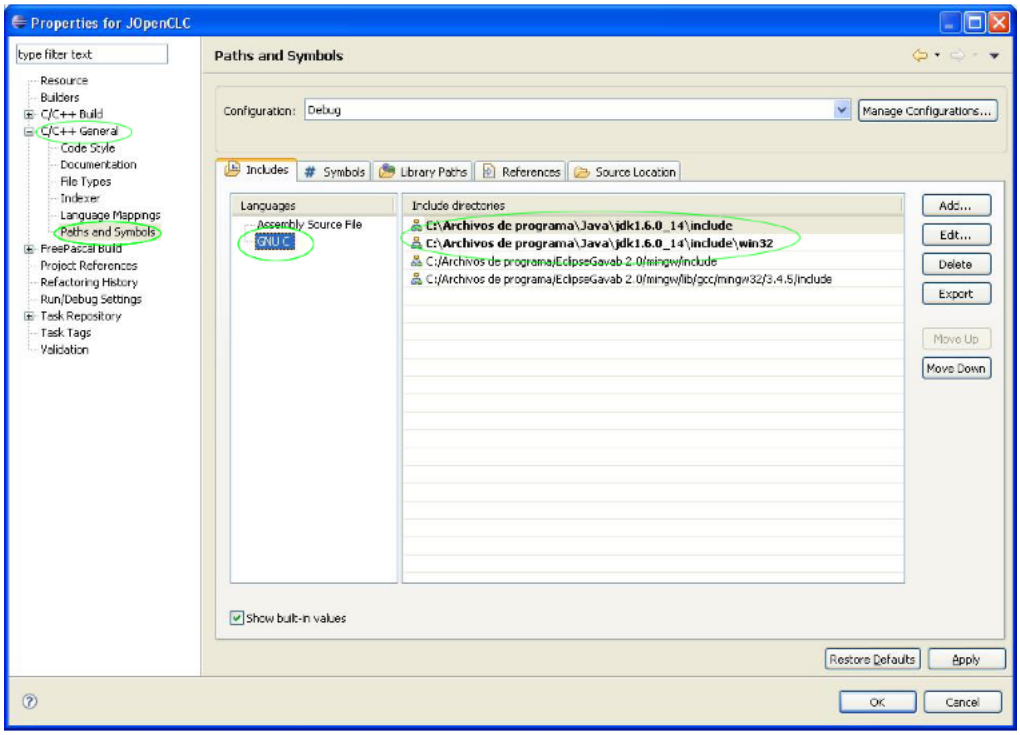

Figura 7.1: Directorios de inclusión necesarios

Incluir librerías *OpenCL*

Es necesario que el compilador sea capaz de enlazar las librerías de *OpenCL* al generar el proyecto, por lo que en la configuración del proyecto, dentro de *C/C++ Build* → *Settings*, en la pestaña *Tool Settings*, dentro del apartado *MinGW C Linker* → *Libraries*, debemos realizar dos cambios. Primero, en la parte superior, en *Libraries (-l)*, es necesario añadir la librería *OpenCL32*, y en la parte inferior, en *Library search path (-L)*, será necesario añadir el directorio donde se encuentre dicha librería. En la Figura 7.2 aparecen reflejados estos cambios a realizar.

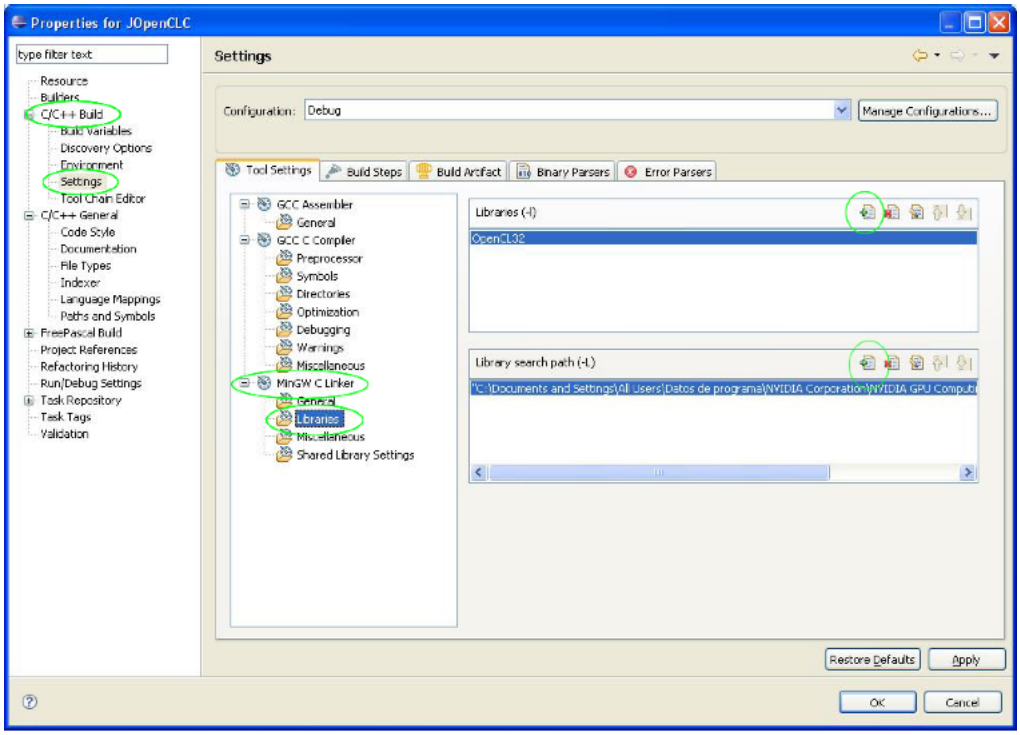

Figura 7.2: Inclusión de librería OpenCL

Declaración de variables dentro de un bucle *for*

Aunque no forma parte estrictamente de *JavaOpenCL*, si está función no está activa, no será posible compilar el proyecto. Para activarla, dentro de la configuración del proyecto, en el apartado *C/C++ Build* → *Settings* → *Miscellaneous* será necesario añadir en el apartado *Other flags* la línea *-std=c99*.

Símbolos necesarios

Es necesario definir un nuevo símbolo dentro del proyecto. Para ello, en la configuración del proyecto, dentro de *C/C++ Build* → *Settings* → *GCC C Compiler* → *Symbols*, en el apartado *Defined symbols (-D)* se añade el símbolo *_JNI_IMPLEMENTATION_*. La Figura 7.3 muestra el resultado de esta operación.

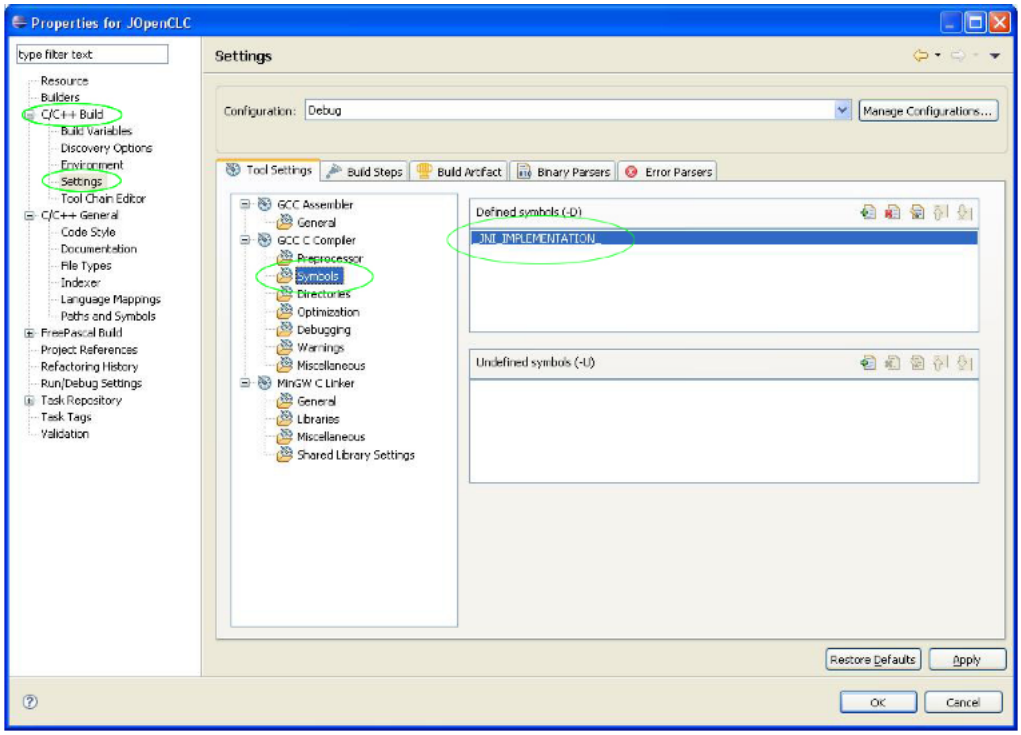

Figura 7.3: Añadir un nuevo símbolo

Opciones del *linker*

Por último, el *linker* necesita tener activados algunos *flags* para realizar la generación del proyecto correctamente. Estos *flags* se añaden en la configuración del proyecto, dentro del apartado *C/C++ Build* → *Settings* → *MinGW Linker* → *Miscellaneous*, en la opción *Linker flags* será necesario añadir la siguiente línea: `-Wl, -kill-at`. La Figura 7.4 muestra el resultado de esta configuración.

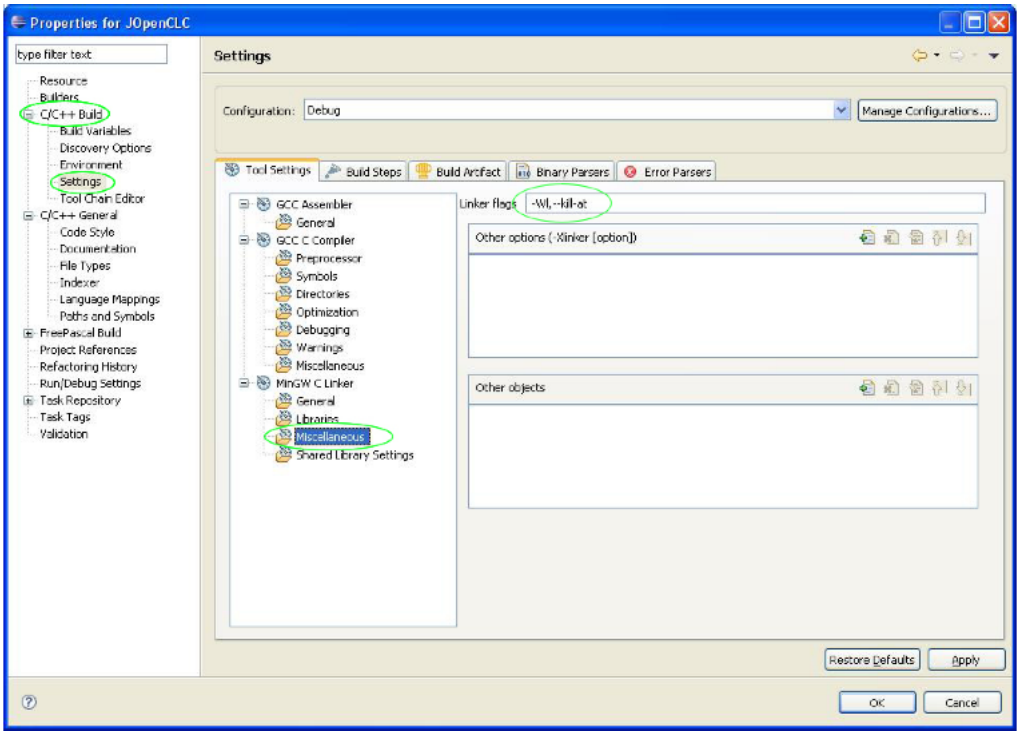

Figura 7.4: Añadir *flags* al *linker*

Bibliografía

[1] Introducción a la computación paralela ww.saber.ula.ve/bitstream/123456789/15969/1/com_par.pdf.

[2] Jocl - http://www.jocl.org.

[3] Open toolkit library - http://www.opentk.com.

[4] Opencl - http://www.khronos.org/opencl.

[5] Python::opencl - http://python-opencl.next-touch.com.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K.Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. *The Landscape of Parallel Computing Research: A View from Berkelev.* Technical report, EECS Department, University of California, Berkeley, 2006.

[7] O. Chafik. Nativelibs4java - http://code.google.com/p/nativelibs4java/.

[8] S. Dámaris, A. Calderón, and J. C. V. Rebaza. Metodologías Ágiles. page 37, 2007.

[9] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and Uri C. Weiser. *Many Core vs. Many Thread Machines: Stay away from the valley.*

[10] F. L. Hernández. Introducción a jni. *MacProgramadores*, 2007.

[11] Project Kenai. http://kenai.com/projects/jocl/pages/home.

[12] Khronos Group. *OpenCL Overview.*

[13] A. Munshi. Especificación opencl 1.0
http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf
, Octubre 2009.