Trabajo de Fin de Grado

Extendiendo Paralldroid para la Generación Automática de código OpenCL

Extending Paralldroid for the Automatic Generation of OpenCL code

Autor: Sergio Manuel Afonso Fumero

La Laguna, 9 de junio de 2015

D. **Francisco Carmelo Almeida Rodríguez**, con N.I.F. 42.831.571-M Catedrático de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Alejandro Acosta Díaz**, con N.I.F. 78.852.786-T, como cotutor

C E R T I F I C A N

Que la presente memoria titulada:

"*Extendiendo Paralldroid para la Generación Automática de código OpenCL.*"

ha sido realizada bajo su dirección por D. **Sergio Manuel Afonso Fumero**, con N.I.F. 45.866.473-B.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 9 de junio de 2015

Agradecimientos

A Francisco Almeida y Alejandro Acosta, por la ayuda que me han prestado en todo momento y por todo lo que he podido aprender de ellos.

A mi familia, por apoyarme cada vez que lo he necesitado y porque sin su apoyo y confianza no habría llegado hasta aquí.

Licencia

Resumen

El crecimiento sin precedentes que vive el mercado de los Smartphones también ha llevado consigo el desarrollo de múltiples arquitecturas diferentes en constante evolución. En la actualidad sigue siendo un reto el aprovechamiento de las capacidades computacionales de estas diversas arquitecturas, ya que su heterogeneidad requiere un conocimiento muy específico sobre las mismas, lo que supone un aprendizaje muy lento para el desarrollador de aplicaciones de propósito general.

Paralldroid es un programa desarrollado por el Grupo de Computación de Altas Prestaciones de la Universidad de La Laguna cuyo objetivo es facilitar a los desarrolladores de aplicaciones móviles, concretamente de Android, el aprovechamiento de las crecientes capacidades computacionales de estos dispositivos.

Paralldroid añade un conjunto de anotaciones al lenguaje Java que se utilizan para marcar clases, métodos y variables de tal forma que se puede generar código paralelo. Las anotaciones añadidas se basan en las definidas por el estándar OpenMP 4.0, pero han sido adaptadas al paradigma de programación orientada a objetos.

El desarrollador sólo tiene que escribir una aplicación orientada a objetos en la que indique mediante anotaciones las partes del código que desea optimizar. Esto ofrece la posibilidad de obtener un buen rendimiento sin incurrir en un coste de desarrollo elevado.

El objetivo de este trabajo ha sido el desarrollo de una extensión de Paralldroid para la generación de código nativo y OpenCL para la ejecución de código paralelo en la GPU.

Palabras clave: Android, Paralldroid, OpenCL, transformación fuente-a-fuente.

Abstract

The unprecedented growth that the handheld systems market (smartphones, tablets, ...) is living has also induced the development of multiple different architectures that are in constant evolution. Currently it is still a challenge to take advantage of the computational capabilities of all these architectures, because the heterogeneity that exists requires a very specific knowledge about them, which implies a high learning curve for general purpose programmers.

Paralldroid is a program developed by the High Performance Computing Group of the University of La Laguna, which goal is to ease the efficient use of the increasing computational capacities of Android mobile devices to general purpose application developers.

Paralldroid adds a set of annotations to the Java language that are used to mark classes, methods and variables in a way such that parallel code may be generated from it. These annotations are based in the OpenMP 4.0 specification, but they are adapted to the object oriented programming paradigm.

The developer just implements an object oriented Java application and introduces a set of Paralldroid annotations in the sections of code to be optimized. This offers the possibility of obtaining a good performance without incurring in high development costs.

The goal of this work is the development of an extension of Paralldroid for the generation of native and OpenCL code for the execution of parallel code in the GPU.

Keywords: *Android, Paralldroid, OpenCL, source-to-source transformation.*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Antecedentes y estado del arte

La evolución de muchas de las tecnologías ubícuitas actuales ha sido posible debido al desarrollo de las tecnologías de Sistemas en Chip (SOCs). La era de las tecnologías de la información, por su parte, ha empujado una revolución de las comunicaciones globales. Como resultado de esta revolución, la potencia de cómputo de los dispositivos móviles se ha incrementado. Las tecnologías disponibles en los sistemas de escritorio ahora están implementados en sistemas embebidos y móviles. En este escenario, podemos encontrar que nuevos procesadores que integran arquitecturas multinúcleo, GPUs y DSPs están siendo desarrollados para este mercado. Nvidia Tegra, Qualcomm Snapdragon y Samsung Exynos son plataformas que van en esta dirección.

En cuanto al desarrollo de software, muchos frameworks han sido desarrollados para dar soporte a la creación de programas para estos dispositivos. Las principales compañías en este mercado tienen sus propias plataformas: Windows Phone de Microsoft, iOS de Apple y Android de Google son rivales en el mercado de los smartphones. El desarrollo de aplicaciones para estos dispositivos ahora es más sencillo. Además del problema de crear hardware energéticamente eficiente, nos hemos encontrado con la difícil tarea de crear programas eficientes y mantenibles que se ejecuten en éste.

Dado el alto nivel de heterogeneidad que existe entre estos dispositivos es obligatoria la creación de herramientas para que el desarrollo de los mismos siga siendo mantenible en términos de programabilidad. En este escenario, encontramos una fuerte diferenciación entre los desarrolladores de software móvil tradicionales y los programadores paralelos. Los primeros tienden a utilizar frameworks de desarrollo de alto nivel para la creación de programas, sin ningún conocimiento de programación paralela (Android: Eclipse/Android Studio + Java, Windows Phone: Visual Studio + C#, iOS: XCode + Objective C), mientras que los segundos están acostumbrados a trabajar en Linux, escribiendo sus

programas directamente en OpenCL a más bajo nivel. Los primeros se aprovechan de la expresividad de los lenguajes de alto nivel mientras los segundos afrontan los retos de la programación de altas prestaciones. Paralldroid ayuda a acercar estos dos mundos.

Paralldroid es un framework de desarrollo que permite la implementación automática de código nativo o Renderscript para dispositivos móviles (Smartphones, tablets...). El desarrollador escribe código secuencial en un lenguaje de alto nivel al que añade anotaciones, de forma que ciertas secciones se pueden implementar en código nativo o Renderscript. Paralldroid utiliza la información proporcionada por estas anotaciones para generar un nuevo programa que incorpora las secciones de código a ejecutar en la CPU o GPU.

1.2. Alcance del proyecto

Se parte de que Paralldroid permite la generación tanto de código nativo en C como de código Renderscript paralelo que se ejecuta en la GPU.

En este proyecto se pretende llevar a cabo el desarrollo de una extensión de Paralldroid para la generación de código OpenCL a partir de código Java con anotaciones.

Las tareas que conforman este proyecto son las siguientes:

- Desarrollar una clase de traducción para modificar el código Java escrito por el usuario. El objetivo del código generado es mantener la interfaz del código original pero, de forma transparente al usuario, delegar su funcionalidad al código OpenCL generado.
- Escribir un nuevo traductor para el código nativo generado. Este código tiene como objetivo intermediar en la comunicación bidireccional Java-OpenCL.
- Crear un nuevo traductor para el código OpenCL C paralelo a partir del código anotado escrito por el usuario.
- Hacer pruebas para comprobar el correcto funcionamiento del código generado.
- Llevar a cabo tests de rendimiento para comparar la ejecución de algoritmos iguales utilizando el código generado con los distintos traductores de Paralldroid.

1.3. Fases del desarrollo

En la tabla 1.1 se presenta un cronograma con las tareas que se ha realizado para completar los objetivos propuestos en este trabajo:

Tarea	**Duración**
Instalar y configurar el Android SDK y NDK y ejecutar algunas aplicaciones de prueba incluidas	5 horas
Instalar el SDK de CUDA y escribir varios algoritmos paralelos en OpenCL sobre vectores, matrices e imágenes	25 horas
Leer las publicaciones sobre Paralldroid y estudiar el significado de cada una de las anotaciones que define	2 horas
Estudiar la Interfaz Nativa de Java (JNI) y hacer pruebas de desarrollo de aplicaciones nativas en Android	5 horas
Familiarización con la arquitectura de clases del código de OpenJDK y Paralldroid	20 horas
Creación de las plantillas de las clases de traducción de código Java y nativo para OpenCL a partir de las clases para la generación de código nativo	2 horas
Adaptación de Paralldroid para utilizar los nuevos traductores cuando el *Target* es OpenCL	1 hora
Implementación de la generación básica de código Java, nativo y OpenCL C para casos de uso simples de Paralldroid	150 horas
Desarrollo de una aplicación Android de pruebas que hace uso de la generación OpenCL de Paralldroid para realizar operaciones sobre arrays en paralelo	3 horas
Modificación de la generación de código nativo para soportar la creación de varias instancias de las clases generadas	15 horas
Implementación de la notificación de errores OpenCL desde el código nativo mediante el uso de un nuevo tipo de excepciones	3 horas
Solución de los problemas existentes al hacer subclases de las clases de traducción y refactorización general de los traductores	15 horas
Adición de algoritmos de procesamiento de imágenes en paralelo a la aplicación de testing de OpenCL desarrollada previamente	2 horas
Implementación del soporte de uso de los métodos de la clase Math de Java en los kernels OpenCL, mediante la traducción a su función equivalente	1 hora
Adición de soporte a la traducción de variables estáticas en la generación de código nativo y OpenCL	20 horas
Modificación de las clases de traducción para permitir el uso de los parámetros opcionales de ciertas anotaciones	5 horas

Tarea	**Duración**
Implementación de la inicialización de variables estáticas desde el método *initJNI* y de la traducción de los métodos *Declare* al código nativo y al código OpenCL C de forma independiente	5 horas
Redacción de la documentación del código escrito	15 horas
Pruebas y benchmarks del código OpenCL generado y el código Renderscript y Java equivalentes sobre algoritmos de procesamiento de imágenes	2 horas

Tabla 1.1: Cronograma de tareas

Capítulo 2

El modelo de desarrollo en Android

Android es un sistema operativo basado en Linux diseñado principalmente para dispositivos móviles como smartphones y tablets. Las aplicaciones Android están escritas en Java y su Kit de Desarrollo de Software (SDK) proporciona las librerías y herramientas necesarias para construir, probar y debuguear aplicaciones Android.

A pesar de que el desarrollo para Android se lleva a cabo mayoritariamente en Java, también existen otros modelos en Android que permiten llevar a cabo el desarrollo de aplicaciones para este sistema operativo. En la figura 2.1 podemos observar el proceso de compilación de estos distintos modelos.

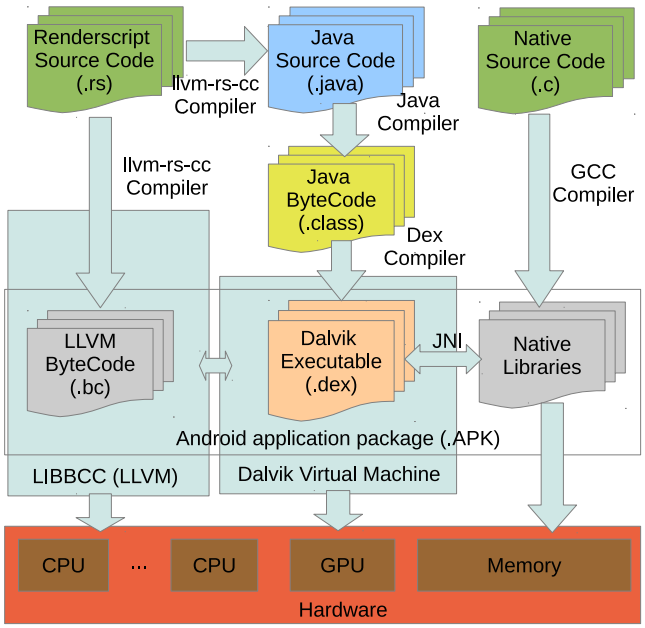


Figura 2.1: El modelo de desarrollo en Android

2.1. Modelo tradicional

Tal como se mencionó anteriormente, el modo más frecuente en el que se lleva a cabo el desarrollo de aplicaciones Android es en el lenguaje de programación Java. En la parte central de la figura 2.1 se puede observar el conjunto de estados por los que pasa el código hasta finalmente ejecutarse en el dispositivo.

El desarrollador de aplicaciones móviles escribe el código en Java que es convertido por el compilador de Java en bytecode almacenado en ficheros `.class`. Este bytecode es posteriormente adaptado a la máquina virtual Dalvik mediante la creación de ficheros `.dex`.

Dalvik es la máquina virtual de Java que se encuentra en Android y que se encarga de gestionar los recursos del dispositivo (memoria, almacenamiento, CPU, GPU, energía...) a través del kernel de Linux.

Los ficheros `.dex` son colocados en el `.apk` de la aplicación Android y son finalmente ejecutados por Dalvik cuando el usuario utiliza la misma.

A partir de Android 5.0, Dalvik ha sido reemplazado por Android Runtime (ART), que sustituye la interpretación del código Java por compilación a código nativo de los programas a la hora de instalarlos. Esto permite obtener un mejor rendimiento a costa de un mayor consumo de espacio de almacenamiento y tiempo de instalación. Aún así, desde el punto de vista del programador el funcionamiento es el mismo, puesto que el contenido de los `.apk` es igual, ya que ART es compatible con los ficheros `.dex`.

2.2. Renderscript

Para la explotación de las capacidades computacionales en dispositivos actuales, Android proporciona Renderscript. Se trata de una API de computación de altas prestaciones que define un lenguaje basado en el estándar de C99.

Renderscript permite la ejecución de aplicaciones paralelas en multitud de procesadores como pueden ser la CPU, GPU o DSPs, gestionando de manera automática la distribución del trabajo a través de los núcleos de cómputo disponibles en el dispositivo.

El modelo de compilación y ejecución de Renderscript está representado en la parte izquierda de la figura 2.1. Además del código Java de la aplicación, el programador escribe distintos métodos en Renderscript que son los que se aprovechan de las características previamente comentadas. Estos ficheros Renderscript (`.rs`) son compilados por LLVM, un compilador basado en Clang, produciendo clases Java que envuelven el código Renderscript y bytecode LLVM (`.bc`) que terminan en el `.apk` de la aplicación.

Renderscript es útil para mejorar el rendimiento de algoritmos de procesamiento de imágenes, modelado matemático u otro tipo de operaciones que requieren mucha computación matemática.

2.3. Nativo

Android proporciona herramientas y librerías para el desarrollo de aplicaciones nativas mediante el Kit de Desarrollo Nativo (NDK). Éste permite implementar partes de la aplicación que se ejecuta en la máquina virtual Dalvik usando lenguajes nativos como C o C++.

Para comunicar el código Java y nativo se utiliza la Interfaz Nativa de Java (JNI), que permite que la implementación de los métodos de una clase sea especificada en código nativo, y proporciona una librería nativa que define una interfaz para poder obtener y modificar datos y ejecutar código Java. Así, facilita una comunicación bidireccional entre Java y C/C++.

El método utilizado para la compilación de aplicaciones que hacen uso del NDK, tal como muestra la parte derecha de la figura 2.1, consiste en compilar por separado el código Java y nativo. El código Java utiliza el procedimiento explicado en el apartado 2.1, mientras que el código nativo es compilado por el compilador de GNU (GCC) en forma de librería compartida (`.so`) que en tiempo de ejecución es enlazada mediante JNI al resto de la aplicación Java.

El NDK está pensado para operaciones que utilizan de manera intensiva la CPU y no utilizan mucha memoria, como procesamiento de señales y simulaciones físicas, o para reutilizar código ya escrito en un lenguaje nativo, pero su utilización no siempre supone un incremento del rendimiento, por lo que no se recomienda su uso para esto, puesto que lo que sí que siempre supone es un incremento en la complejidad de la aplicación.

Sin embargo, la ejecución de código nativo abre las puertas a la utilización de librerías de bajo nivel que permitan acceder a los recursos computacionales de los dispositivos móviles fuera de los métodos convencionales impulsados por Android, como es el caso de OpenCL.

Capítulo 3

Paralldroid

Paralldroid ha sido diseñado para facilitar el desarrollo de aplicaciones paralelas en plataformas Android. Se presupone que las plataformas móviles estarán formadas por una CPU clásica y algún otro tipo de coprocesador como una GPU que pueda ser explotada mediante alguna librería de computación de altas prestaciones, como pueden ser Renderscript u OpenCL.

Para ello, Paralldroid define una serie de anotaciones que, aplicadas al código Java de una aplicación, pueden convertirse en código de algún otro de los modelos de programación en Android e incluso paralelizarse, consiguiendo así un rendimiento significativamente mejor sin por ello incurrir en un coste de desarrollo significativamente mayor. En definitiva, permite gestionar todos los distintos modelos de programación de Android de manera automática mientras el programador de aplicaciones tan sólo trabaja en código Java con el que ya está familiarizado, tal como se puede observar en la figura 3.1.

Las anotaciones que define Paralldroid están basadas en la especificación de OpenMP 4.0, que incluye directivas para aceleradores. Sin embargo, las directivas de OpenMP, a pesar de poder utilizarse en lenguajes de programación orientados a objetos como C++, no están bien adaptadas al paradigma de programación orientado a objetos. Como el objetivo es que las anotaciones sean utilizadas en Java, y Java es un lenguaje orientado a objetos, para que el uso de las mismas sea lo más intuitivo posible, éstas se han adaptado para integrarse mejor en este paradigma de programación.

3.1. Anotaciones

Las anotaciones definidas por Paralldroid son las que se indican a continuación. En el apéndice A.1 se puede ver un ejemplo en el que se hace uso de las mismas.

- **@Target**: Esta directiva crea un entorno de datos en el dispositivo. Hace que los campos de la clase en la que se aplica sean mapeados al entorno

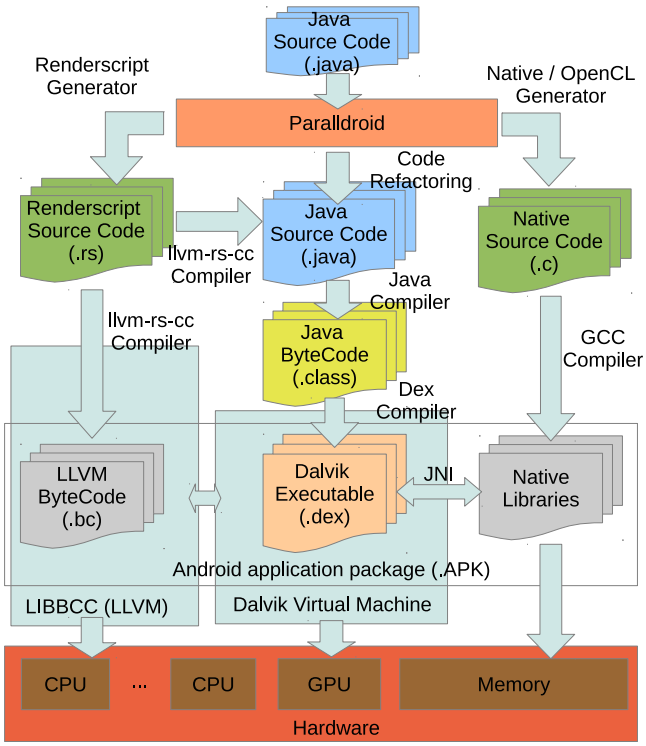


Figura 3.1: El modelo de desarrollo en Paralldroid

de destino y que sus métodos sean ejecutados en el mismo. Siempre se aplica a las definiciones de clases y posee un parámetro para indicar el tipo de entorno que se quiere crear (Nativo, Renderscript u OpenCL). Esta directiva decide los traductores que Paralldroid utiliza para traducir una clase.

- **@Map**: Su función es mapear una variable Java al entorno de destino de la clase. Se puede aplicar a campos de la clase y a parámetros de métodos y recibe un parámetro para indicar el tipo de mapeo: *Alloc*, *To*, *ToFrom* o *From*. Para los campos de la clase anotados con *@Map*, Paralldroid genera getters y setters dependiendo del tipo de mapeo.

- **@Declare**: Esta anotación especifica que el elemento tiene que declararse en el contexto del dispositivo, por lo que sólo es accesible en el mismo. Se puede aplicar tanto a campos de la clase como a métodos. Recibe un parámetro opcional para campos de tipo array en el que se puede indicar una expresión para inicializarlo con ese tamaño.

- **@Parallel**: Se aplica sólo a métodos e indica que el método se debe ejecutar en paralelo. Se permite su uso en contextos Renderscript y OpenCL, pero no en el contexto nativo.

- **@Input y @Output**: Especifican los vectores de entrada y salida en la ejecución de un método paralelo. Se aplican sobre parámetros de tipo

array. Sólo están soportados en Renderscript.

- **@NumThreads**: Se utiliza para definir la variable que indica en un método paralelo cuántos hilos deben ejecutarse. Se puede aplicar al propio método si se le indica el parámetro `field`, lo que significa que utiliza uno de los campos de la clase, o se puede aplicar a uno de los parámetros del método. En cualquier caso, sólo puede haber una anotación *@NumThreads* en cada método paralelo. Si la variable es de un tipo entero, su valor se utiliza como número de hilos, mientras que si se trata de un array, su longitud pasa a ser el número de hilos.

- **@Index**: Permite especificar la variable utilizada como índice en cada uno de los hilos que ejecutan un método paralelo. Se aplica a parámetros de tipo entero en métodos paralelos. Su valor se asigna en tiempo de ejecución.

Para que las directivas *@Map*, *@Declare* y *@Parallel* surtan algún efecto, deben encontrarse en una clase anotada como *@Target*, al igual que las anotaciones *@Input*, *@Output*, *@NumThreads* e *@Index* sólo funcionan si están asociadas a un método *@Parallel*.

3.2. Generación de código

Paralldroid hace uso del parser de Java de OpenJDK para obtener el Árbol Sintáctico Abstracto (AST) del código que escribe el usuario. Una vez hecho esto, el procedimiento que utiliza para generar código es crear nuevos ASTs para cada uno de los lenguajes de destino dependiendo de las anotaciones utilizadas y del código escrito por el usuario y transformar estos nuevos ASTs en código. En la figura 3.2 se representa este proceso de manera gráfica.

Utilizando el detector de anotaciones, analiza el AST de Java buscando la anotación *@Target* en cada una de las clases definidas. Cuando no se encuentra dicha anotación, el código del usuario no es modificado ni analizado. Cuando se detecta la anotación en una clase, se utiliza el valor de su parámetro para determinar el conjunto de traductores que se utilizará para analizar el código de la clase y generar un nuevo AST.

Para cualquiera de los lenguajes siempre se modifica el código Java que el usuario escribió originalmente para actuar como interfaz entre el resto del programa y el contexto del dispositivo donde se ejecutará el código. Por ello se encarga de inicializar el contexto del dispositivo, transferir los datos desde y hacia el mismo, lanzar la ejecución de código y liberar los recursos cuando el recolector de basura de Java elimina cada instancia.

Dependiendo del lenguaje de destino, se utilizará el `NativeTreeTranslator`, el `RSTreeTranslator` o una combinación entre el `OCLTreeTranslator` y el `OCLKernelTreeTranslator` para procesar el AST Java original y generar el AST que representa el código equivalente en alguno de los otros lenguajes de programación, atendiendo a las anotaciones que éste tuviera.

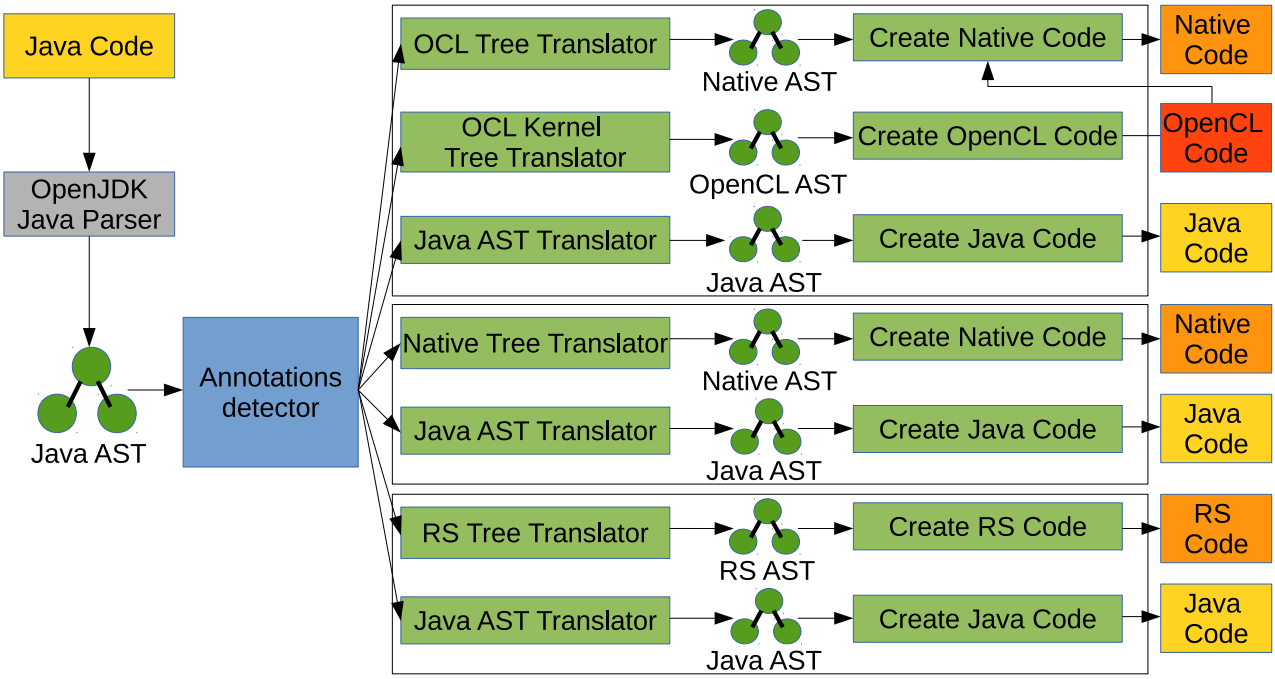


Figura 3.2: Proceso de transformación de ASTs en Paralldroid

Capítulo 4

Desarrollo

4.1. Aportaciones

En esta sección se detallará el conjunto de funcionalidades y mejoras que han sido añadidas a Paralldroid como resultado de la ejecución de este proyecto.

Ha sido creada la clase de traducción que permite convertir métodos Java marcados como paralelos en kernels OpenCL C que pueden ser ejecutados por la GPU del dispositivo móvil en paralelo. Los métodos `@Declare` también son traducidos mediante este traductor para crear las funciones de apoyo que pueden ser llamadas desde los kernels. Se permite además el uso de la mayoría de funciones matemáticas proporcionadas por la clase `Math` de Java a través de su traducción a la función equivalente en OpenCL C.

Este traductor no es independiente en el sentido de que el AST que genera no va directamente a un fichero, puesto que no sería una modificación sencilla el añadir a OpenJDK la posibilidad de generar varios ASTs a partir de la traducción de un solo AST de origen, sino que es llamado desde el traductor de código nativo de OpenCL y el código resultante se almacena en una cadena de texto en el código nativo que se compila en tiempo de ejecución mediante la librería nativa de OpenCL.

También se ha creado la clase de traducción que genera el código nativo que permite comunicar la aplicación Java con el contexto OpenCL. Este traductor ha sido desarrollado a partir del traductor nativo que ya había sido desarrollado para Paralldroid previamente, pero se le ha añadido gran cantidad de funcionalidades, como lo son:

- Creación del contexto y la cola de comandos de OpenCL al inicializar la primera instancia de la clase. También se compila el código OpenCL C correspondiente a los métodos `@Parallel` y `@Declare` en tiempo de ejecución. Por otro lado se crea una función nativa que es llamada al eliminar la última instancia de la clase por el recolector de basura, la cual

se encarga de liberar los objetos de OpenCL compartidos por todas las instancias de la clase (programa, kernels, cola de comandos y contexto).

- Comprobación del código de error devuelto por cada llamada a funciones de la API de OpenCL y construcción y lanzamiento de una excepción con la información sobre el fichero, la línea y el código de error devuelto para permitir al usuario gestionar estos problemas desde su código Java.

- Creación de buffers en el contexto OpenCL que se mantienen sincronizados mediante los getters y setters de la aplicación del usuario.

- Inicialización de las variables de instancia de la clase en un método nativo que actúa como constructor y liberación de las mismas en otro método que es llamado al ser eliminada la instancia por el recolector de basura de Java.

- Sustitución del código de los métodos paralelos por el código que configura la llamada a un kernel de OpenCL y lo lanza, una vez los parámetros de entrada han sido enviados al dispositivo. Cuando la computación termina, los parámetros de salida de la función son actualizados con los valores leídos de nuevo desde el contexto OpenCL. Se evita la obtención de los datos sobre los campos de la clase al terminar la ejecución del kernel para evitar transferencias de memoria en la medida de lo posible, y porque los getters ya se encargan de actualizar la memoria antes de devolverla el código Java.

Además de esto, la generación de código nativo fue modificada para solucionar algunos problemas que presentaba previamente y para añadir nuevas funciones. Todos estos cambios también afectan a la generación de código nativo de OpenCL, ya que se trata de una extensión del traductor nativo. Algunos de los cambios más importantes introducidos fueron:

- Soporte para la coexistencia de varias instancias de la clase. La implementación que existía para la generación de código nativo utilizaba variables globales para almacenar los campos de la clase, lo cual significaba que éstas se compartían entre instancias de la clase. Esto se modificó para almacenar en cada instancia un puntero a los datos nativos correspondientes a los campos no estáticos de la clase y ahí almacenar las variables nativas correspondientes.

- Soporte para uso de variables estáticas. Se definió un método `initJNI` que recibe como parámetros la lista de campos estáticos inicializados de la clase y que se implementa en el contexto nativo.

Se añadió una clase para la modificación del AST de Java para que el nuevo código delegue la implementación de los métodos al código nativo generado. Este traductor es casi idéntico al traductor de código Java para el `@Target` nativo. Sin embargo, el traductor de Java para el acceso al código nativo fue modificado también para implementar las funciones mencionadas anteriormente: la posibilidad de creación de varias instancias de la clase y el soporte para variables estáticas. Esto se consiguió mediante un contador de instancias y un nuevo campo de la clase que almacena un puntero a los datos nativos. En el apéndice A.1 hay un ejemplo en el que el funcionamiento de esto se puede ver con mayor detalle.

Además de las nuevas funcionalidades ya mencionadas, se realizó un trabajo de refactorización del código bastante importante para permitir el reaprovechamiento de ciertas funcionalidades en las clases que permiten crear nuevos nodos del AST (makers), que son las que se utilizan en los traductores para ir cambiando el AST original a medida que lo recorren.

4.2. Problemas encontrados

En este apartado se expondrá una serie de problemas encontrados a la hora de llevar a cabo este proyecto, indicando la solución por la que se ha optado en aquellos problemas que se ha podido resolver.

Tamaño del proyecto

El primer problema encontrado fue el tamaño del proyecto. Paralldroid había sido desarrollado durante varios años antes de comenzar este proyecto, y ha seguido siendo desarrollado mientras este proyecto se llevaba a cabo.

Por este motivo el tamaño del proyecto, incluyendo aplicaciones de prueba y el propio compilador, es suficientemente grande como para que contribuir en él sea problemático inicialmente.

Hubo que hacer dos reuniones con el desarrollador principal de Paralldroid (cotutor de este proyecto) para comprender la arquitectura de clases del compilador y para obtener una idea general del modo en el que se trabaja para hacer transformaciones en los ASTs. Después de esto hizo falta dedicar una semana a analizar el código de Paralldroid antes de comenzar a hacer contribuciones.

Localización del código de los kernels

Inicialmente la idea era generar un fichero diferente con el código de cada uno de los métodos paralelos de la clase traducidos a OpenCL C (kernels), pero OpenJDK está diseñado de tal forma que cada AST se traduce a un solo fichero.

El problema es que en cada clase puede haber múltiples métodos anotados como paralelos, pero habría que colocarlos en el mismo fichero porque en otro caso habría que hacer modificaciones sustanciales al código de OpenJDK.

Se optó por implementar parcialmente la clase de traducción de kernels, que los ASTs generados fueran convertidos a código dentro del traductor de código nativo y este código se colocara dentro de una cadena de texto en el programa nativo generado.

La implementación parcial del traductor consistía en que en lugar de poder recibir el AST de la clase Java anotada, tan sólo tiene soporte para recibir definiciones de métodos de la clase, que es lo que puede traducir.

Con esta aproximación inicialmente se creaba una cadena de texto en el código nativo para cada uno de los métodos paralelos de la clase. En cada cadena se encontraba tanto el código del correspondiente kernel como el código de cada uno de los métodos `@Declare`.

El problema con esto es que había mucho código repetido en las cadenas de texto (todos los métodos `@Declare`), lo que hacía crecer mucho el tamaño del programa nativo. Para solucionar esto se consiguió generar código nativo que permitía la existencia de todos los kernels en el mismo programa (la misma cadena de texto) y que cada uno se compilara en un objeto diferente mediante el compilador en tiempo de ejecución de OpenCL.

Esta nueva aproximación, además de solucionar el problema para el que fue tomada, solucionó el problema inicial porque ya no haría falta generar más de un fichero con código OpenCL para cada clase. Sin embargo, como el traductor fue desarrollado parcialmente, habría que rediseñarlo para que funcione de manera independiente al traductor nativo.

Debugging del código nativo y OpenCL

El SDK de Android proporciona herramientas para el debugging de aplicaciones escritas en Java. Éste permite ejecutar paso a paso, inspeccionar y modificar el valor de cada variable en tiempo de ejecución, etc.

Sin embargo, para hacer lo mismo con el código nativo, el NDK de Android no proporciona el mismo tipo de facilidades, por lo que encontrar fallos en este código es mucho más complicado.

Si el debugging del código nativo escrito para la aplicación móvil ya es muy complicado, puesto que prácticamente la única facilidad que se proporciona es la posibilidad de escribir al log del sistema, el debugging del código OpenCL lo es mucho más.

Para paliar en la medida de lo posible la incapacidad para conocer lo que sucede en el contexto OpenCL al ejecutar métodos paralelos se implementó el sistema de notificación de errores de OpenCL mediante excepciones específicas.

Aún así, la información que proporcionan estas excepciones es muy limitado porque sólo indican la línea en el código nativo donde se detectó el error y el código de error OpenCL que sucedió. Para problemas de compilación del código OpenCL se debería modificar el mensaje de error para que sea el registro de la compilación del programa y así obtener algo de información más útil.

Gestión de los recursos compartidos entre instancias

El problema era que sería altamente ineficiente inicializar OpenCL por separado en cada una de las instancias que se hacen de la clase, por lo que había que buscar alguna forma en la que sólo hubiera que hacerlo una vez para cada clase.

JNI define una función que se llama al inicializar la librería y otra que se llama al liberarla. Implementando la creación y liberación de variables de OpenCL en este par de funciones se obtendrían las funcionalidades deseadas.

Sin embargo, no sólo el contexto de OpenCL se comparte entre instancias de la clase, sino que también son compartidos los campos estáticos de la clase y algunas referencias globales. Los parámetros de las funciones ya descritas de inicialización y liberación de la librería JNI no permitirían acceder a los campos estáticos de la clase Java a la que está asociada esta librería, por lo que estas funciones no podrían utilizarse.

En lugar de eso, se genera un nuevo método estático privado en la clase Java llamado `initJNI` que se implementa en código nativo. Este método recibe la lista de campos estáticos inicializados de la clase como parámetros y en su implementación se gestiona tanto la creación del contexto OpenCL, como la creación de las referencias globales y la inicialización de las variables globales nativas con los parámetros recibidos. También se crea un nuevo método `releaseJNI` con la funcionalidad inversa para liberar la memoria reservada por `initJNI`.

Para conseguir que `initJNI` sólo se llamara al crear la primera instancia de la clase y que `releaseJNI` sólo se llamara al liberar la última hubo que implementar un contador de instancias en la clase Java. Esto, sin embargo, impide la utilización de métodos estáticos para acceder a los campos estáticos de la clase, porque tiene que haber al menos una instancia para que los accesos a dichas variables sean válidos.

Traducción de métodos sin anotaciones

Los métodos sin anotaciones tienen la particularidad de que se traducen a código nativo, puesto que para traducirse a OpenCL C tendrían que ser paralelos o funciones de apoyo a métodos paralelos, pero en el segundo caso no podrían ser llamados desde Java.

A pesar de que pudiera parecer que estos métodos no deberían suponer un problema, lo cierto es que el hecho de que pueden acceder a campos de la clase hace que surjan problemas de sincronización entre los datos en el contexto OpenCL y nativo.

El código nativo generado realiza la transferencia de buffers de memoria de campos de la clase hacia el contexto OpenCL en los métodos de inicialización de instancias, de inicialización de la clase y en los setters, y realiza la transferencia inversa en los getters. Esto implica que las transferencias de memoria entre CPU y GPU sólo se realizan cuando el código Java de la aplicación necesita enviar o recibir estos datos.

El problema que surge es que desde que se asigna un valor a un buffer de memoria hasta que se hace una llamada a un método nativo no se sabe si sus datos han sido modificados en el dispositivo por alguna llamada a un kernel.

Por este motivo, se implementó el análisis del código de cada uno de estos métodos para determinar los campos de la clase de tipo array a los que acceden (los arrays se mapean a objetos de memoria de OpenCL). Una vez determinados los objetos de memoria potencialmente leídos o modificados por la función nativa, se añadió la obtención de los datos del contexto OpenCL de dichos objetos antes del cuerpo de la función escrito por el usuario, y se añadió el código de escritura de vuelta al dispositivo antes de cada uno de los puntos de salida de la misma (sentencias `return` o el final del cuerpo de la función).

Traducción de métodos `@Declare`

Por definición, se trata de métodos que se definen en el contexto de destino de la generación de código. Sin embargo, en la práctica nos encontramos que estos métodos pueden ser llamados desde métodos paralelos u otros métodos `@Declare`, pero también pueden ser llamados desde métodos no anotados.

Como ya se ha comentado, los métodos paralelos se deben definir en el contexto OpenCL, pero los métodos no anotados se definen en el contexto nativo. Para que ambos tipos de método puedan acceder a las funciones `@Declare` hay que generarlas tanto como funciones OpenCL C de soporte a los kernels como funciones nativas de apoyo no accesibles desde Java.

Hay que asegurar, pues, que las traducciones tienen la misma semántica en los dos casos, pero el código generado es diferente puesto que son lenguajes ligeramente distintos pero, sobre todo, porque la forma de acceder a los campos de la clase es diferente y porque las funciones matemáticas disponibles en los dos contextos son algo distintas.

Modificación de campos de la clase en los kernels

Para que los kernels tuvieran acceso a los campos de la clase, se optó por pasarlos como parámetros además de los parámetros que los kernels tuvieran por indicación del usuario.

Sin embargo, esto introduce un problema conocido para el cual todavía no se ha encontrado solución. El problema consiste en la modificación de variables simples que representan campos de la clase. Cuando se modifica algún elemento de un array, eventualmente estas modificaciones vuelven al código nativo y a la aplicación Java, pero como las variables simples se pasan por copia, ninguna de las modificaciones que se realicen a estas variables en el cuerpo de un método paralelo se verán reflejadas en el código nativo o Java, ni tampoco en el resto de kernels o en futuras ejecuciones del mismo método.

Capítulo 5

Pruebas de rendimiento

5.1. Metodología

Las pruebas de rendimiento se han llevado a cabo en el dispositivo *Odroid-XU3* sobre varios algoritmos de procesamiento de imágenes. Las especificaciones del dispositivo son las siguientes:

- Procesador: Samsung Exynos 5422 Octa.
- CPUs: ARM® Cortex™-A15 Quad 2GHz y Cortex™-A7 Quad 1.3GHz.
- Memoria: 2GB LPDDR3 RAM a 933 MHz.
- GPU: ARM® Mali™-T628 MP6.

El *Odroid-XU3* es una placa de reducido tamaño que posee todas las funcionalidades que puede proporcionar un ordenador de sobremesa. Por otro lado, este pequeño dispositivo permite la ejecución de varios sistemas operativos modernos basados en Linux, como Ubuntu 14.04 o Android 4.4. Tiene una arquitectura basada en ARM y sus principales características son un buen rendimiento computacional encapsulado en un pequeño factor de forma con una alta eficiencia energética.

En este caso las pruebas fueron realizadas en el sistema operativo Android 4.4 (KitKat).

Los algoritmos sobre los que las pruebas fueron realizadas son:

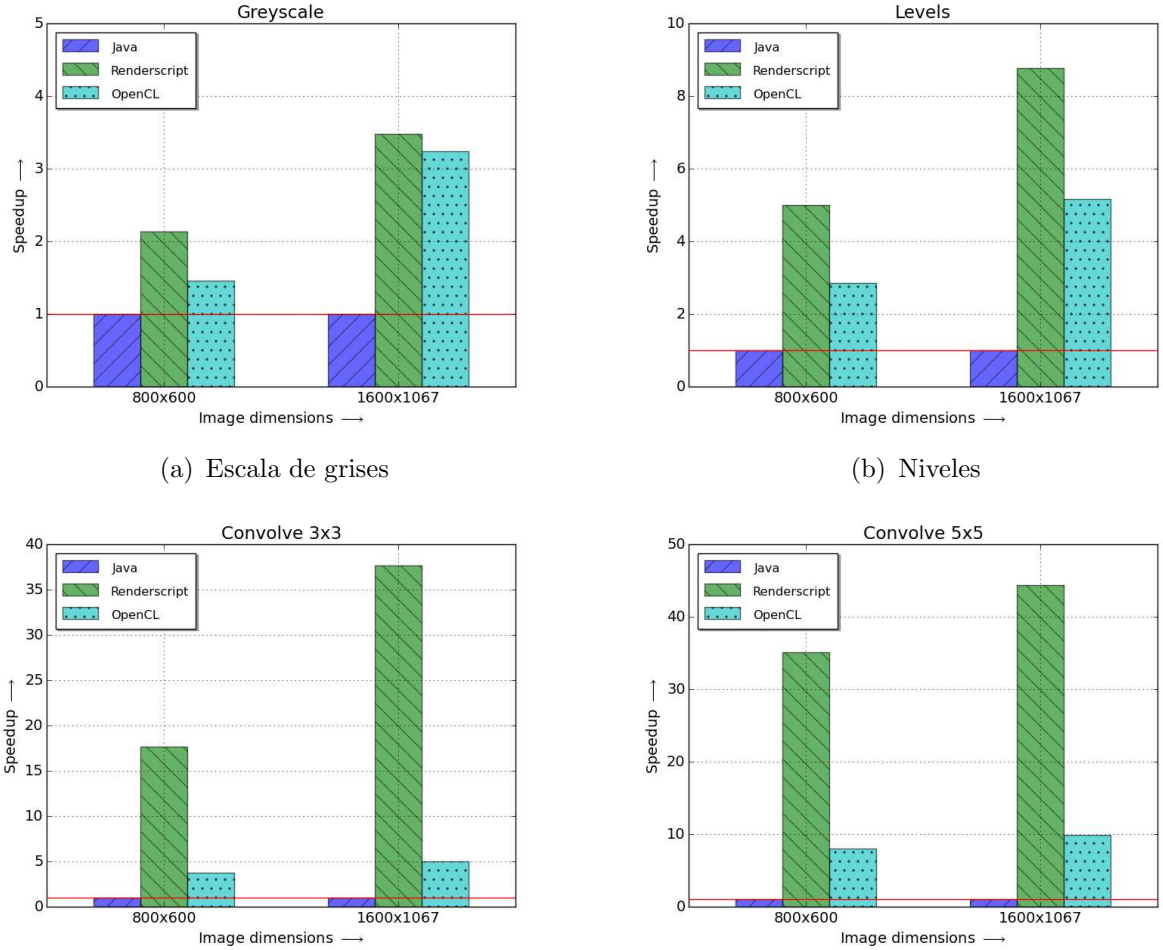
- Escala de grises: Transformación de una imagen en color a escala de grises.
- Niveles: Modificación de los niveles de brillo, contraste y color de una imagen.
- Convolución 3x3: Aplicación de un filtro lineal en el que el valor de cada píxel es una media ponderada del mismo con sus 8 píxeles vecinos.

- Convolución 5x5: Aplicación de una convolución utilizando una máscara de 5x5 píxeles; es decir, haciendo la media ponderada con los 24 píxeles vecinos.

Todos estos algoritmos han sido ejecutados sobre dos imágenes de distinto tamaño (800×600 y 1600×1067) y utilizando una implementación manual en Java y las implementaciones auto-generadas por Paralldroid para Renderscript y OpenCL a partir de una misma implementación en Java con anotaciones.

5.2. Resultados

Los resultados de los tests de rendimiento se presentan en la figura 5.1. Ahí se representa la mejora de rendimiento que supone la ejecución del código Renderscript y OpenCL generado por Paralldroid con respecto a la implementación secuencial en Java de los mismos, atendiendo al tiempo de ejecución de cada una de las versiones del algoritmo.



(a) Escala de grises

(b) Niveles

(c) Convolución 3x3

(d) Convolución 5x5

Figura 5.1: Resultados de las pruebas de rendimiento

Se puede observar con claridad el aumento de rendimiento que se obtiene al implementar los algoritmos en paralelo frente a la implementación secuencial en Java. También se puede ver que cuanto mayor es la cantidad de procesamiento que se lleva a cabo, más notable es la mejora de rendimiento.

Sin embargo, también es notable la diferencia de rendimiento que existe entre Renderscript y OpenCL, a pesar de que ambos ejecutan las tareas en paralelo. Hay varios motivos que pueden explicar esta discordancia:

- Renderscript distribuye las tareas entre la CPU y GPU, aprovechando todos los recursos del dispositivo, mientras que la implementación de OpenCL realizada se limita al uso de la GPU. Las GPUs en dispositivos móviles aún son tecnología muy novedosa, por lo que es posible que la ejecución en CPUs a día de hoy sea más rápida.

- La generación de OpenCL en Paralldroid todavía es muy básica, ya que no realiza ningún tipo de optimizaciones al código del usuario, aparte de que su uso no está oficialmente soportado en Android. Por otro lado, la generación de Renderscript ya ha sido muy trabajada en Paralldroid y su ejecución está optimizada en Android.

- Desde que los datos son enviados por la aplicación Java hasta que llegan al contexto OpenCL deben pasar por dos transformaciones y copias (Java $\leftrightarrow$ Nativo y Nativo $\leftrightarrow$ OpenCL), mientras que en Renderscript este proceso se maneja de forma casi transparente, posiblemente de una manera más optimizada.

- La inicialización y liberación del contexto de OpenCL se llevan a cabo, respectivamente, al crear la primera instancia de la clase y al eliminar la última. A la hora de realizar los tests se ha trabajado con una instancia cada vez y, como el recolector de basura es impredecible, cabe la posibilidad de que, al realizar los tests, en cada una de las iteraciones se inicializara y liberara OpenCL, afectando negativamente al rendimiento medido.

- Renderscript puede trabajar directamente con objetos de tipo `Bitmap` y las operaciones que realiza sobre los píxeles se aprovechan de instrucciones vectoriales. En OpenCL hay que transformar el `Bitmap` en un array de píxeles y luego obtener y procesar por separado cada una de las componentes (RGBA) de cada píxel. El uso de instrucciones vectoriales por sí solo supone un aumento de rendimiento importante que se observa con claridad en los resultados de los tests.

A pesar de la diferencia de rendimiento entre la implementación Renderscript y OpenCL de los algoritmos, hay que tener en cuenta también que la implementación OpenCL no es para nada lenta, puesto que la velocidad de ejecución

con respecto al código Java equivalente llega a ser hasta 10 veces mejor en el ejemplo 5.1(d).

Capítulo 6

Conclusiones y líneas futuras

Se ha presentado Paralldroid, un framework que simplifica la generación automática de código nativo, Renderscript u OpenCL en dispositivos Android. El usuario anota la definición de clases Java secuenciales y Paralldroid transforma automáticamente este código en una implementación en alguno de los lenguajes de destino.

La generación de código OpenCL presentada en este trabajo demuestra conseguir buenas mejoras de rendimiento, con respecto al código Java secuencial, en la ejecución de algoritmos de procesamiento de imágenes fácilmente paralelizables.

Sin embargo, todavía hay mucho lugar para mejoras, dada la gran diferencia de rendimiento que existe entre el código Renderscript y OpenCL generados. No obstante hay que tener en cuenta que OpenCL no está soportado oficialmente en Android, y que, en principio, es más recomendable seguir mejorando la generación de Renderscript para esta plataforma. Aún así, puede ser interesante en cualquier caso la generación de OpenCL en caso de que Paralldroid se extienda en un futuro para su integración con otras plataformas de desarrollo.

Como continuación a este trabajo cabría la posibilidad de añadir soporte para tipos de datos vectoriales, uso de objetos de tipo `Bitmap` o arrays multidimensionales. Para comprobar su correcto funcionamiento podría contemplarse la posibilidad de probar la generación de código OpenCL con ejemplos más complejos que hagan uso de más características de Paralldroid.

Capítulo 7

Summary and Conclusions

Paralldroid has been presented, a framework that simplifies the automatic generation of Native C, Renderscript and OpenCL code on Android devices. The user annotates the definition of sequential Java classes and Paralldroid automatically turns this code into its implementation in one of its supported destination languages.

The OpenCL code generation presented in this paper has demonstrated to achieve good performance improvements, regarding to its sequential Java counterpart, in the execution of easily parallelizable image processing algorithms.

Nevertheless, there is still a lot of room for improvement, given the big difference in performance that exists between the generated Renderscript and OpenCL code. However it has to be taken into account the fact that OpenCL is not officially supported in Android, and that, in principle, it is more advisable to keep improving the Renderscript code generation for this platform.

As a continuation to this work it would be possible to add support to vector data types, multidimensional arrays or the use of `Bitmap` objects. In order to ensure the correct generation of OpenCL code, there is the possibility of testing it with more complex examples that make use of more features of Paralldroid.

Capítulo 8

Presupuesto

Elemento	**Cantidad**	**Precio/ud.**	**Descripción**
Odroid-XU3	1	120€	Dispositivo Android donde realizar las pruebas y tests de rendimiento. Tiene soporte para OpenCL 1.1.
Tarjeta de memoria	2	40€	Tarjetas de memoria con sistema operativo preinstalado para su uso en el Odroid.
Cables	1	20€	HDMI, USB 3.0, ...

Tabla 8.1: Presupuesto

Apéndice A

Apéndices

A.1. Generación de código: Escala de grises

Código Java anotado

```
@Target(OPENCL)
public class GrayScale {
  @Declare
  private float gMonoMult[] = {0.299f, 0.587f, 0.114f};

  @Map(TO)
  private int width;

  @Map(TO)
  private int height;

  public GrayScale(int width, int height) {
    this.width = width;
    this.height = height;
  }

  @Parallel
  public void runTest(@NumThreads @Map(TO) int [] srcPxs, @Map(FROM) int [] outPxs, @Index int x) {
    int acc;

    acc = (int)(((srcPxs[x]) & 0xff) * gMonoMult[0]);
    acc += (int)(((srcPxs[x] >> 8 ) & 0xff) * gMonoMult[1]);
    acc += (int)(((srcPxs[x] >> 16) & 0xff) * gMonoMult[2]);

    outPxs[x] = (acc) + (acc << 8) + (acc << 16) + (srcPxs[x] << 24);
  }
}
```

Código Java generado

```
@Target(OPENCL)
public class GrayScale {
  static {
    System.loadLibrary("grayscale");
  }

  private static int instanceCount = 0;
  private long instanceDataPtr;

  @Declare
  private float[] gMonoMult = {0.299F, 0.587F, 0.114F};

  @Map(TO)
  private int width;
  @Map(TO)
  private int height;

  public GrayScale(int width, int height) {
    this.width = width;
    this.height = height;
    if (instanceCount == 0) initJNI();
    ++instanceCount;
    initGrayScale(gMonoMult, width, height);
  }

  @Parallel
  public native void runTest(@NumThreads @Map(TO) int[] srcPxsrunTest, @Map(FROM) int[] outPxsrunTest);

  public native void setWidth(int width);
  public native void setHeight(int height);

  private native void initGrayScale(float[] gMonoMult, int width, int height);
  private native void destroyGrayScale();

  private static native void initJNI();
  private static native void releaseJNI();

  protected void finalize() {
    destroyGrayScale();
    --instanceCount;
    if (instanceCount == 0) releaseJNI();
  }
}
```

Código nativo generado

```
#include <CL/cl.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <jni.h>
#include <stdbool.h>

typedef struct InstanceData {
  float* gMonoMult;
  jfloatArray gMonoMultGrayScale;
  size_t sz_gMonoMult;
  cl_mem gMonoMult_OCL;
  int width;
  int height;
} InstanceData;

jclass currentClass = NULL;
jclass openCLExceptionClass = NULL;

cl_int err = CL_SUCCESS;
cl_platform_id platform = NULL;
cl_device_id device = NULL;
cl_context context = NULL;
cl_command_queue queue = NULL;
const char* program_src =
  "void __kernel runTest(__global float* m_gMonoMult, int m_width, int m_height,"
  "__global int* srcPxs, __global int* outPxs) {"
  " int x = get_global_id(0);"
  " int acc;"
  " acc = (int)(((srcPxs[x]) & 255) * m_gMonoMult[0]);"
  " acc += (int)(((srcPxs[x] >> 8) & 255) * m_gMonoMult[1]);"
  " acc += (int)(((srcPxs[x] >> 16) & 255) * m_gMonoMult[2]);"
  " outPxs[x] = (acc) + (acc << 8) + (acc << 16) + (srcPxs[x] << 24);"
  "}";
cl_program program = NULL;
cl_kernel kernel_runTest = NULL;

// Funciones helpers "getInstanceData", "defaultExceptionText", "throwOpenCLError", ...
...

JNIEXPORT void JNICALL Java_com_example_vectoroperations_GrayScale_runTest(JNIEnv *env,
    jobject obj, jintArray srcPxsrunTest, jintArray outPxsrunTest) {
  InstanceData* self = getInstanceData(env, obj);
  // Copiar los parámetros de tipo array (jintArray) en arrays nativos (int*)
  int* srcPxs = (*env)->GetIntArrayElements(env, srcPxsrunTest, 0);
  ...
  // Crear los buffers OpenCL y enviar el array @Map(TO) al contexto OpenCL
  size_t sz_srcPxs = (*env)->GetArrayLength(env, srcPxsrunTest) * sizeof(int);
  cl_mem srcPxs_OCL = clCreateBuffer(context, CL_MEM_READ_ONLY, sz_srcPxs, NULL, &err);
  if (err != CL_SUCCESS) throwOpenCLError(env, defaultExceptionText(__LINE__));
  err = clEnqueueWriteBuffer(queue, srcPxs_OCL, CL_TRUE, 0, sz_srcPxs, srcPxs, 0, NULL, NULL);
  if (err != CL_SUCCESS) throwOpenCLError(env, defaultExceptionText(__LINE__));
  ...
  // Configurar la lista de parámetros del lanzamiento del kernel
  err = clSetKernelArg(kernel_runTest, 0, sizeof(self->gMonoMult_OCL), &self->gMonoMult_OCL);
  if (err != CL_SUCCESS) throwOpenCLError(env, defaultExceptionText(__LINE__));
  err = clSetKernelArg(kernel_runTest, 1, sizeof(self->width), &self->width);
```

```
  ...
  // Lanzar el kernel
  err = clEnqueueNDRangeKernel(...);
  ...
  // Obtener los datos de tipo array @Map(FROM) del contexto OpenCL
  err = clEnqueueReadBuffer(queue, outPxs_OCL, CL_TRUE, 0, sz_outPxs, outPxs, 0, NULL, NULL);
  ...
  // Copiar los contenidos del array nativo @Map(FROM) al correspondiente en Java y liberar
  // memoria
  clReleaseMemObject(srcPxs_OCL);
  (*env)->ReleaseIntArrayElements(env, srcPxsrunTest, srcPxs, JNI_ABORT);
  ...
}
JNIEXPORT void JNICALL Java_com_example_vectoroperations_GrayScale_initJNI(JNIEnv *env) {
  // Inicialización de referencias globales
  jclass cls = (*env)->FindClass(env, "es/ull/pcg/paralldroid/api/OpenCLException");
  openCLExceptionClass = (*env)->NewGlobalRef(env, cls);
  ...
  // Inicialización de OpenCL
  err = clGetPlatformIDs(1, &platform, NULL);
  if (err != CL_SUCCESS) throwOpenCLError(env, defaultExceptionText(__LINE__));
  ...
}
JNIEXPORT void JNICALL Java_com_example_vectoroperations_GrayScale_releaseJNI(JNIEnv *env) {
  // Liberación de referencias globales y OpenCL
  ...
}
JNIEXPORT void JNICALL Java_com_example_vectoroperations_GrayScale_initGrayScale(JNIEnv *env,
    jobject obj, jfloatArray gMonoMultGrayScale_PARAM, int width_PARAM, int height_PARAM) {
  // Creación del objeto que contiene los datos de instancia y almacenamiento del
  // puntero en la instancia para poder recuperarlo en el resto de métodos
  InstanceData* self = malloc(sizeof(InstanceData));
  jfieldID bufferField = getInstanceDataField(env);
  (*env)->SetLongField(env, obj, bufferField, (long)self);
  // Inicialización de variables de instancia y creación de buffers OpenCL
  ...
}
JNIEXPORT void JNICALL Java_com_example_vectoroperations_GrayScale_destroyGrayScale(JNIEnv *env,
    jobject obj) {
  // Liberación de los arrays en los datos de instancia y del objeto "self"
}
JNIEXPORT void JNICALL Java_com_example_vectoroperations_GrayScale_setWidth(JNIEnv *env,
    jobject obj, int width_PARAM) {
  InstanceData* self = getInstanceData(env, obj);
  self->width = width_PARAM;
}
JNIEXPORT void JNICALL Java_com_example_vectoroperations_GrayScale_setHeight(JNIEnv *env,
    jobject obj, int height_PARAM) {
  InstanceData* self = getInstanceData(env, obj);
  self->height = height_PARAM;
}
```

Bibliografía

[1] Alejandro Acosta and Francisco Almeida. Towards an unified heterogeneous development model in android. In *Eleventh International Workshop HeteroPar'2013: Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, 2013.

[2] Anandtech. AMD Outlines HSA Roadmap: Unified Memory for CPU/GPU in 2013, HSA GPUs in 2014. `http://www.anandtech.com/show/5493/`.

[3] Apple. iOS: Apple mobile operating system. `http://www.apple.com/ios`.

[4] Google. Android mobile platform. `http://www.android.com`.

[5] Microsoft. Windows Phone: Microsoft mobile operating system. `http://www.microsoft.com/windowsphone`.

[6] Nvidia. GPUDirect Technology. `http://developer.nvidia.com/gpudirect`.

[7] OpenMP. OpenMP application program interface, version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[8] OpenMP. The OpenMP API specification for parallel programming. `http://openmp.org/wp/openmp-specifications/`.