Escuela de Doctorado
y Estudios de Posgrado
Universidad de La Laguna

Trabajo de Fin de Máster

Computación en Paralelo y Dispositivos Móviles

Parallel Computing and Mobile Devices

Paula Elena Expósito Estévez

La Laguna, 7 de julio de 2023

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. **Vicente José Blanco Pérez**, con N.I.F. 42.171.808-C profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A N

Que la presente memoria titulada:

"*Computación en Paralelo y Dispositivos Móviles*"

ha sido realizada bajo su dirección por Dña. **Paula Elena Expósito Estévez**, con N.I.F. 43.382.565-B.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 7 de julio de 2023

Agradecimientos

En este trabajo me gustaría agradecer especialmente a las personas que me han adentrado en el mundo del paralelismo, las altas prestaciones y los compiladores.

A Vicente, que es quien me animó en un inicio a meterme en este fregado, y que es un ejemplo de trabajo y curiosidad constante.

A Paco, que es quien me ha guiado en este proyecto, y que al final resultó ser uno de esos extraños profesores que sí que saben hacer cosas.

Y a Sergio, por la ayuda que me ha prestado y por todo lo que aprendido de él, tanto a través de conversaciones como a través de su trabajo, que es mucho.

Reconocimientos

Este trabajo ha sido posible gracias al Ministerio de Ciencia e Innovación a través del Programa Estatal de I+D+i Orientada a los Retos de la Sociedad RTI tipo B numero PID2019-107228RB-I00.

Licencia

Resumen

La capacidad de los sistemas heterogéneos, como los System-on-Chip *de los dispositivos móviles, para realizar Computación de Altas Prestaciones trae consigo la necesidad de disponer de herramientas que permitan explotar las posibilidades que estos ofrecen. Sin embargo, la dificultad de desarrollar aplicaciones basadas en modelos de programación paralelos ha impedido que estas capacidades se puedan aprovechar.*

Por este motivo, en este trabajo se presenta Tradux*, un traductor fuente a fuente que a partir de código Java anotado genera código paralelo más eficiente en C/C++ y OpenCL. El código generado puede ejecutarse tanto en la CPU como en la GPU de entornos Android y Linux. Este proyecto parte de Paralldroid y Fancier, dos aplicaciones desarrolladas por el Grupo de Computación de Altas Prestaciones de la Universidad de Laguna con el mismo objetivo. El análisis de rendimiento realizado demuestra que con esta herramienta se consigue mejorar el rendimiento al poder ejecutar los programas en paralelo. Además, la librería se ha diseñado pensando en la usabilidad por parte del usuario y del programador, facilitando el uso del código generado por parte del primero y permitiendo el mantenimiento y la incorporación de nuevas funcionalidades de forma sencilla por parte del segundo.*

Palabras clave: Traducción fuente a fuente, Dispositivos móviles, Aceleración del hardware, Sistemas heterogéneos, Programación paralela, Análisis de rendimiento

Abstract

Mobile devices' Systems-on-Chip brings the capacity to make High-Performance Computing on heterogeneous systems. Therefore we need tools that exploit these capabilities. However, the difficulty of programming applications based on parallel programming models has prevented this from happening.

For this reason, we present Tradux in this project. It is a source-to-source translator which generates more efficient parallel code in Java, C/C++ and OpenCL from annotated Java code. The generated code can be executed on the CPU or the GPU of Android and Linux platforms. This project has its basis on Paralldroid and Fancier, two applications developed by the High-Performance Computing Group of the University of La Laguna with the same objective. The performance analysis shows that Tradux improves the efficiency of the programs by executing them in parallel. In addition, this library has been designed to ease the use of the generated code by the user and to facilitate its maintenance and extension by the programmer in the future.

Keywords: Source-to-source translator, Mobile devices, Hardware acceleration, Heterogeneous systems, Parallel programming, Performance analysis

Índice general

Índice de Figuras

Índice de Tablas

Listings

Capítulo 1

Introducción

En este primer capítulo se presentan la motivación y los objetivos principales del proyecto, así como la metodología utilizada para su desarrollo.

1.1. Motivación

Los dispositivos móviles pueden ser usados para realizar Computación de Altas Prestaciones debido a que el hardware ya incorpora arquitecturas paralelas heterogéneas, es decir, que combinan diferentes tipos de procesadores en un mismo chip. Sin embargo, a pesar de que el rendimiento de los componentes hardware ha mejorado mucho en los últimos años, el software no ha evolucionado a la misma velocidad. Por lo que el potencial de estos procesadores suele estar desaprovechado. Esto ocurre debido a que la programación de aplicaciones aceleradas en estos dispositivos no está estandarizada y a que el desarrollo de software basado en modelos de programación paralelos supone una gran dificultad para los programadores.

1.2. Objetivo

El objetivo de este trabajo es implementar una herramienta que facilite a los desarrolladores el uso eficiente del potencial de los procesadores especializados de los *System-on-Chip*. Concretamente nos centramos en las GPUs y en dispositivos móviles basados en el sistema operativo Android. Se parte de Paralldroid, un traductor de código fuente a fuente que genera código que se puede ejecutar en las GPUs de los móviles. Sin embargo, debido a problemas en su diseño, que han condicionado el desarrollo de nuevas funcionalidades, se ha optado por desarrollar una nueva librería que lo reemplace.

1.3. Metodología

Las tareas que se llevaron a cabo a lo largo del proyecto fueron las siguientes:

- Análisis y estudio de la librería Fancier.

- Instalación y configuración del entorno de desarrollo.
- Análisis de las aplicaciones de prueba de Fancier.
- Análisis y estudio de Paralldroid.
- Análisis de otras aplicaciones del campo.
- Análisis de las mejoras, cambios e integración de Fancier y Paralldroid.
- Generación manual de la traducción objetivo.
- Implementación del backend Fancier Native.
- Implementación del backend Fancier Java.
- Implementación del backend Fancier OpenCL.
- Implementación de una interfaz común.
- Implementación de las aplicaciones de prueba.
- Análisis de la librería Rancid.
- Ejecución de las pruebas de rendimiento.
- Interpretación de los resultados de las pruebas.

1.4. Resultados

Finalmente, se ha conseguido desarrollar la librería Tradux partiendo de las librerías Paralldroid y Fancier. Con Tradux se consigue traducir código Java anotado a lenguajes más eficientes como C y OpenCL en los sistemas operativos Android y Linux. Además de ser una herramienta que fácilmente se puede extender para adaptarla a nuevas plataformas y lenguajes, modificar el código para implementar nuevas funcionalidades requiere menos esfuerzo que hacerlo en Paralldroid.

Capítulo 2

Antecedentes y estado del arte

En este capítulo se describen los antecedentes y el estado del arte de la ejecución de programas acelerados en arquitecturas heterogéneas basadas en el sistema operativo Android. Para ello se expone el modelo de desarrollo de Android y se explican las librerías Paralldroid [1] y Fancier [3].

2.1. Estado del arte

El uso y el desarrollo de dispositivos basados en arquitecturas heterogéneas de bajo consumo se ha visto significativamente incrementado en las últimas décadas, y es de esperar que en los próximos años este incremento continúe. Los avances en el desarrollo de sistemas **System-on-Chip**, en adelante **SoC**, donde se integran diferentes tipos de procesadores como **CPUs multinúcleo**, **GPUs** o **DSPs** aumentan la capacidad de cómputo y permiten diversificar el ámbito de aplicación de estos dispositivos.

Cada uno de los aceleradores nombrados anteriormente tiene su propia interfaz y un modelo de programación diferente, un modelo de programación que es paralelo y específico para cada arquitectura. Sin embargo, la dificultad para utilizar modelos de programación paralelos y tener que utilizar uno diferente para cada acelerador ha hecho que las mejoras de rendimiento se hayan limitado, en su mayoría, al hardware y no al software, por lo que mucho potencial no se está aprovechando.

Para programar en estos dispositivos se utilizan lenguajes de bajo nivel como CUDA, OpenCL o RenderScript, aunque este último está ya obsoleto. Mientras que en la programación de dispositivos móviles se usan lenguajes de alto nivel. En **Android** se utilizan Java y Kotlin y en **iOS**, C# y Swift. Otras alternativas más tradicionales para ejecutar programas en paralelo son el paradigma de paso de mensajes con MPI, el uso de memoria compartida con OpenMP o incluso OpenACC para ejecutar en las GPUs. El problema de este tipo de tecnologías es que no están disponibles de forma nativa para las plataformas móviles y que tampoco hay un estándar definido.

Existen algunas librerías que facilitan la ejecución de programas paralelos en

Java, principalmente se basan en dos estrategias: traducir código fuente a fuente en tiempo de compilación, Paralldroid y ParallelME, o modificar el *bytecode* en tiempo de ejecución para ejecutar un kernel de OpenCL, Aparapi y TornadoVM. Aunque debido a restricciones de la máquina virtual de Android esta segunda opción no funciona en los dispositivos móviles.

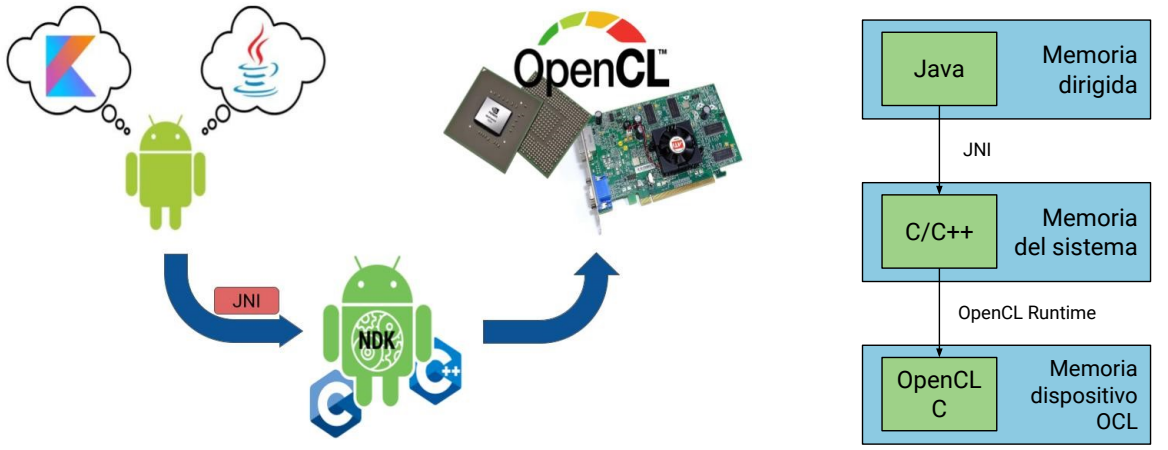
2.2. Modelo de desarrollo en Android

El modelo de ejecución en Android no es trivial, lo que hace que ejecutar código en las GPUs de los dispositivos móviles no sea fácil. A priori nos encontramos con que las aplicaciones en Android se programan con **Java** o con **Kotlin**, ambos se compilan a *bytecode* de Java, que se ejecuta en la máquina virtual de Android con el Android Runtime (ART).

Mientras, los aceleradores sólo aceptan unos pocos lenguajes especializados, entre los que no está Java. **OpenCL** es un estándar que permite desarrollar código para diferentes procesadores como CPUs, GPUs, DSPs y FPGAs. Incluye una API que permite gestionar desde la CPU principal los espacios de memoria, inicializar los parámetros de la ejecución paralela, la compilación del código para la aceleración, etc. Esta API es accesible solamente por programas C/C++.

Por tanto, además del código Java y OpenCL, también hace falta código **C/C++** que haga de intermediario entre ellos. Desde OpenCL no hay problema puesto que el propio lenguaje tiene como requisito ser usado desde C/C++, pero para interconectar el código Java hace falta utilizar la **Java Native Interface** (JNI).

La utilización y forma en que funcionan estos tres lenguajes conlleva que tres tipos de ejecución con su correspondiente espacio de memoria se utilicen a la vez, lo que puede provocar problemas con la sincronización y comunicación de la memoria. Cosa que lo convierte en un punto crítico del programa. En la Figura 2.1 podemos verlo representado.



(a) Ejecución de aplicaciones. (b) Espacios de memoria.

Figura 2.1: Modelo de desarrollo en Android.

2.3. Paralldroid

Paralldroid [4] [5] es un *framework* de alto nivel desarrollado por el Grupo de Computación de Altas Prestaciones (GCAP) de la Universidad de La Laguna para explotar los componentes de los SoC, como GPUs o DSPs, de una forma sencilla y sin tener que codificar un programa diferente para cada acelerador. Inspirándose en el funcionamiento de OpenMP, Paralldroid, a partir de código secuencial anotado genera código paralelo más eficiente que se puede ejecutar en diferentes plataformas.

En Android se permitía utilizar tres modelos de programación diferentes en el momento en que se concibió Paralldroid; Java, RenderScript y C. Por ello, en un principio se incluyeron dos *backends* en la librería para generar código RenderScript y código C. Ambos son más eficientes que el código Java que suele usarse para programar en Android. Más adelante se incluyó una extensión que permitía generar código OpenCL, por lo que se abrió la puerta a poder utilizar Paralldroid en otras plataformas además de Android.

Paralldroid es un traductor fuente a fuente, donde el usuario escribe un código Java anotado y tras ejecutar el programa se genera el código RenderScript, C u OpenCL. Para utilizar el código generado basta con utilizar la clase generada como si fuese la anotada. La forma en que se dan las instrucciones sobre qué código generar se hace a través de las anotaciones. Por ejemplo, para elegir el lenguaje en que se genera el código se utiliza la anotación @Target. Las anotaciones se incluyen la Tabla 2.1.

Anotación	Aplicada a	Parámetro	Ámbito
@Target	Clases	valor	-
@Map	Campos, parámetros de método	valor	@Target
@Declare	Campos, métodos	campo	@Target
@Parallel	Métodos	-	@Target
@Input	Parámetros de método	-	@Parallel
@Output	Parámetros de método	-	@Parallel
@NumThreads	Métodos, parámetros de método	campo	@Parallel
@Index	Parámetros	-	@Parallel

Tabla 2.1: Anotaciones de Paralldroid.

Para cada clase anotada con @Target se crea un entorno de datos que se encarga de la comunicación entre Java y el modelo de programación elegido. Esto se logra a través de los *getters* y los *setters*, que a su vez, se definen con la anotación @Map y los valores TO, FROM y TOFROM, para diferenciar qué método de acceso debe generarse. El resto de anotaciones se encargan de lo siguiente; @Parallel señala los métodos que se paralelizarán, @NumThreads define el número de hilos, @Index es una variable que contiene el índice en el que se está trabajando en cada hilo, @Input y @Output anotan a los parámetros del método paralelo para saber

si son de lectura o escritura y @Declare acompaña a los métodos que solo existen en contexto generado.

En el Listado 2.1 se encuentra un ejemplo de uso de código anotado con Paralldroid. Se trata del filtro de escala de grises aplicado a una imagen. En este caso queremos generar código OpenCL, tenemos una variable privada que solo existirá en el contexto OpenCL, tendremos dos *setters* y un método paralelo que tendrá dos dimensiones, al tratarse el primer parámetro de una imagen de tipo Bitmap, y los índices de acceso serán *x* e *y*.

```
1  # Contexto en el que se quiere generar el código
2  @Target(value = TargetType.OPENCL)
3  public class Grayscale {
4    # Variable que sólo existirá en el contexto OpenCL
5    @Declare
6    private float gMonoMult[] = { 0.229f, 0.587f, 0.114f };
7
8    # Variables de las que se generá un setter, existirán tanto en
9    # Java como en OpenCL
10   @Map(TO)
11   private int width;
12
13   @Map(TO)
14   private int height;
15
16   public GrayScaleGen (int width, int height) {
17     this.width = width;
18     this.height = height;
19   }
20
21   # Método paralelo con dos párametros
22   @Parallel
23   public void run (@NumThreads @Input Bitmap input,
24                    @Output Bitmap output,
25                    @Index int x, @Index int y) {
26     int acc;
27     int pixel = input.getPixel(x, y);
28     acc = (int) (Color.red(pixel) * gMonoMult[0];
29     acc += (int) (Color.green(pixel) * gMonoMult[1];
30     acc += (int) (Color.blue(pixel) * gMonoMult[2];
31
32     output.setPixel(x, y,
33                     Color.argb(Color.alpha(pixel), acc, acc, acc));
34   }
35 }
```

Listado 2.1: Código anotado para posterizar una imagen en Paralldroid

Respecto a la generación de código se usa OpenJDK 7 para poder modificar el proceso de compilación. Este tiene tres etapas:

- **Parseo y entrada**: parsea los archivos de entrada y crea un *stream* de tokens.
- **Detección de anotaciones**: se determina si los archivos de entrada tienen la anotación @Target, si es así se sigue el proceso de traducción.
- **Análisis y generación**: se analiza el código fuente anotado y se genera el código ejecutable. En esta fase hay varios tipos de traductores y se utilizan unos u otros en función de cuál es el lenguaje objetivo. Todos los programas tendrán una parte Java que puede variar en función del contexto objetivo, todos utilizan el mismo Java AST Translator y el mismo Java Tree Translator. Sin embargo, RenderScript tiene su propio RenderScript AST/Tree Translator, C tiene su Native AST/Tree Translator y OpenCL su OpenCL AST/Tree Translator. En la Figura 2.2 se puede ver representado.

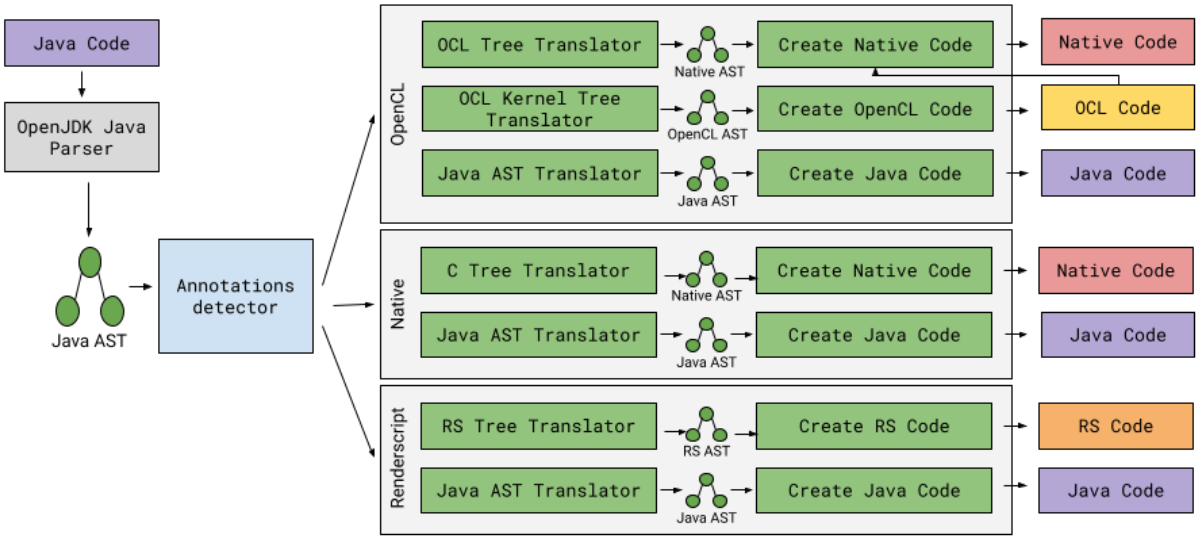


Figura 2.2: Generación de código en Paralldroid.

El código generado por Paralldroid ofrece importantes mejoras de rendimiento a la vez que facilita mucho la producción de código acelerado en estos dispositivos, reduciendo significativamente el esfuerzo de los usuarios. Sin embargo, la metodología de traducción dificulta la introducción de cambios en el código. Además de que algunas de la anotaciones necesarias pueden interpolarse a partir de la implementación de los *kernels* y las posibilidades de traducción son limitadas.

2.4. Fancier

Fancier es una librería de código abierto [6] desarrollada por el GCAP de la Universidad de La Laguna. Esta librería surge de la necesidad de tener un *runtime*

nativo para Paralldroid, que fuese independiente y resolviese eficientemente los problemas de comunicación y sincronización de la memoria.

Se trata de una librería multiplaforma que, al estar orientada a Java, además de poder utilizarse en Android se ha extendido para poder usarse en plataformas Linux. Como no depende de la máquina virtual de Java es fácil de portar. Fancier facilita el desarrollo de aplicaciones aceleradas desde Java al permitir ejecutar código C/C++ y OpenCL. Fancier es una API diseñada en capas que incluye una interfaz común para unificar los tipos de datos, las estructuras y las funciones de Java, C/C++ y OpenCL, tomando como referencia este último. Puede verse en la Figura 2.3. Como observamos, se han mapeado la mayoría de tipos de datos y funciones de OpenCL a C/C++ y a Java, de forma que usando esta API es más sencillo cambiar de un lenguaje a otro.

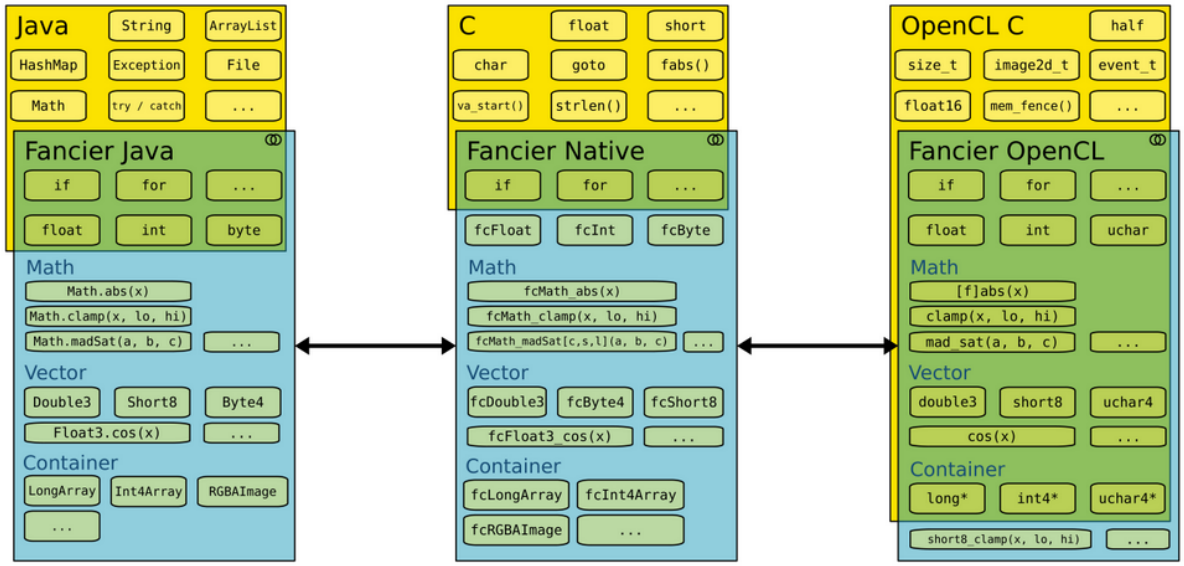


Figura 2.3: Capas de Fancier.

El objetivo principal de Fancier es gestionar de forma automática y eficiente la memoria. Como se comentó en el Capítulo 2.2, al utilizar tres tipos de ejecución diferentes, Java, C/C++ y OpenCL, se utilizan tres espacios de memoria diferente. Lo que usualmente conlleva problemas con la sincronización de los datos cuando se pasa de un espacio a otro y retardos en la comunicación entre estos. Por ello, Fancier aprovecha la memoria unificada de los SoC para reducir los espacios de memoria a dos: la memoria del dispositivo (la GPU en nuestro caso) y la memoria del *host* (la CPU). Sin embargo, dentro de la memoria de la CPU aún tenemos que diferenciar entre la memoria dirigida que utiliza Java y la memoria del sistema que utiliza C/C++. Por ello, Fancier incluye métodos que permiten reservar *buffers* de memoria desde Java, de forma transparente al usuario, para usar directamente la memoria del sistema. En la Figura 2.4 se representa cómo se declara una variable en Fancier y cómo se actualiza un valor. Como vemos, en el segundo ejemplo se llama a `syncToHost`, este método es necesario llamarlo cuando se quiere trabajar en la memoria de la CPU. En el caso de querer utilizar la memoria de la GPU,

porque se quiere ejecutar código en OpenCL, hay que llamar a `syncToDevice` para asegurar que la memoria apunta al dispositivo.

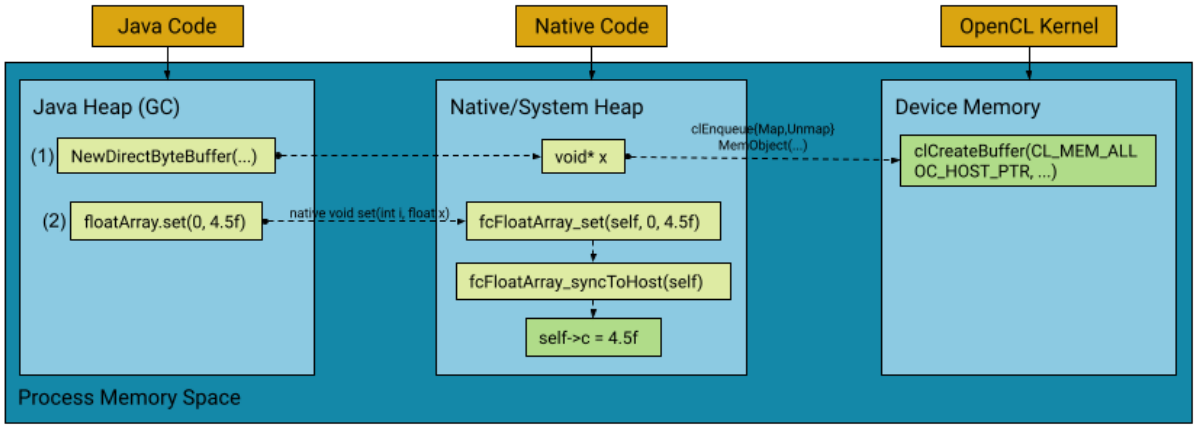


Figura 2.4: Reserva y modificación de un valor de una variable en Fancier.

Capítulo 3

Tradux

En este capítulo se detalla el desarrollo de Tradux. Se explica la necesidad de implementar una nueva herramienta, las mejoras y las limitaciones que presenta, la metodología de desarrollo y de transformación utilizadas y cómo funcionan las pruebas realizadas al código.

3.1. Introducción

Tradux es la librería de alto nivel que se ha diseñado como sucesora de Paralldroid. De esta forma, tiene su mismo objetivo: acelerar aplicaciones móviles a partir de código Java anotado, aunque también puede utilizarse en entornos Linux. Debido a diversos problemas con Paralldroid se concluyó que la mejor opción era implementar una nueva librería desde cero.

Además, Tradux usa Fancier, que se encarga de generar todo el código necesario para inicializar y transferir memoria, crear programas y *kernels* y administrar la cola de ejecución de OpenCL. Además, utiliza su API que unifica las capacidades de Java, C/C++ y OpenCL para que la traducción entre los tres lenguajes sea inmediata. Por cómo está implementado Tradux, de forma modular, pueden incluirse fácilmente *backends* diferentes. Actualmente los que están disponibles son los siguientes:

```
1  import es.ull.hpc.pcg.paralldroid.api.Parallel;
2  import es.ull.hpc.pcg.paralldroid.api.annotations.*;
3
4  public class Foo {
5    @Kernel
6    public void run (@Range FloatArray arr, float kConstant) {
7        int i = (int) Parallel.GlobalID(0);
8        arr.set(i, arr.get(i) + kConstant);
9    }
10 }
```

Listado 3.1: Ejemplo simple en Tradux.

- **Fancier Java**: con este *backend* se traduce el código Java anotado a una versión de código Java de alto rendimiento donde los objetos se reusan y todas las variables se crean al inicio del método, evitando que se reserve la memoria dentro de los bucles. Cuando se utilizan objetos se obtienen los *buffers* y se trabaja con ellos. En el Listado 3.2 podemos ver el código que se genera a partir del código del Listado 3.1.

```
1  public class FooFJ extends Foo {
2    public FooFJ() {
3      super();
4    }
5
6    @Kernel()
7    public void run(@Range() FloatArray arr, float kConstant) {
8      ByteBuffer __paralldroidBuffer_param_arr = arr.getBuffer();
9      int __paralldroid_dims = 1;
10     int[] __paralldroid_index = new int[__paralldroid_dims];
11     int[] __paralldroid_sz = new int[]{(int)arr.length()};
12     for (__paralldroid_index[0] = 0;
13          __paralldroid_index[0] < __paralldroid_sz[0];
14          ++__paralldroid_index[0]) {
15
16       int i = (int)__paralldroid_index[0];
17       FloatArray.setBuffer(__paralldroidBuffer_param_arr, i,
18         FloatArray.getBuffer(__paralldroidBuffer_param_arr, i) +
19         + kConstant);
20     }
21   }
22 }
```

Listado 3.2: Código generado con Fancier Java.

- **Fancier Native**: este *backend* traduce el código anotado a código C, para ello se necesita generar una clase Java y una librería de código C, se generan tanto un `.h` y como un `.c`. En el Listado 3.3 podemos ver la función `run` generada en C equivalente al Listado 3.2.

- **Fancier OpenCL**: con este *backend* se traduce el código anotado para tener el *kernel* en OpenCL y poder ejecutarlo en la GPU. Aquí hace falta generar una clase Java, una librería C y una librería OpenCL. El Listado 3.4 contiene el código generado en OpenCL, equivalente a los Listados 3.2 y 3.3. Como vemos el código es muy similar al original del Listado 3.1.

```
fcError FooFN_run (FooFN* self,
                   fcFloatArray* arr,
                   jfloat kConstant) {

  int __paralldroid_dims = 1;
  int __paralldroid_index[__paralldroid_dims];
  int __paralldroid_sz[] = { arr->len} ;
  fcError err;
  for (__paralldroid_index[0] = 0;
       __paralldroid_index[0] < __paralldroid_sz[0];
       ++__paralldroid_index[0]) {

    jint i = (jint) __paralldroid_index[0];
    err = fcFloatArray_set(arr, i,
                (fcFloatArray_get(arr, i, (&err)) + kConstant));
  }

  return err;
}
```

Listado 3.3: Código generado con Fancier Native.

```
__kernel void run (__global float* arr,
                   int __paralldroidParam_arr_len,
                   float kConstant) {

  int i = (int) get_global_id(0);
  arr[i] = arr[i] + kConstant;
}
```

Listado 3.4: Código generado con Fancier OpenCL.

3.2. Problemática de Paralldroid

A pesar de que Paralldroid es una librería que funciona y consigue acelerar aplicaciones en dispositivos móviles tiene varios problemas que se van a comentar a continuación. En primer lugar, se trata de un proyecto muy grande que se empezó hace más de una década y al que se han ido añadiendo nuevas funcionalidades. A pesar de que se han realizado refactorizaciones a lo largo de estos años el repositorio contiene mucho código, que en su mayoría no está documentado, no se utiliza o no se mantiene, por lo que es difícil orientarse.

El principal problema de Paralldroid es su metodología de transformación. Ésta consiste en tener una clase traductora por cada fichero a generar. Es decir, si se utiliza el *backend* de OpenCL se usan tres clases traductoras; una para la parte Java, otra para el código C y otra para OpenCL. Por tanto, cada *backend* tiene pocas clases `Translator` y es fácil crear nuevos *backends*. En cambio, el proceso de transformación es muy complejo y requiere de miles de líneas de código, por lo que el código de cada una de estas clases es muy difícil de mantener y extender.

A partir de este problema se hacen notar otros, que debido a la dificultad de modificación del código, no se han podido atajar. Por ejemplo, es difícil llevar a cabo grandes cambios en la librería, por ejemplo, no modificar el AST (Árbol Sintáctico Abstracto) de entrada en la traducción, hacer que el traductor de OpenCL sea independiente o permitir trabajar con datos vectoriales.

Por otro lado, hay ramas que se han abandonado, por ejemplo, la generación de código RenderScript ya no se mantiene debido a que el lenguaje, aunque podía ejecutar tanto en la GPU como en la CPU, no garantizaba dónde lo hacía, por tanto, se optó por implementar la extensión de OpenCL. Además, a partir de la versión 12 de Android este lenguaje está obsoleto y no están disponibles sus APIs.

También hay otros elementos problemáticos que se pueden separar en fallos de diseño y en *bugs*. En el primer grupo encontramos elementos como que la localización del código anotado y el generado no está definida, la interfaz pública que codifica el usuario es diferente a la que se genera; por ejemplo, se generan `getters/setters` para todos los atributos de la clase aunque el usuario no los defina, por lo que puede ser confusa de usar. La anotación @Index no aporta nada al usuario y anotaciones como @Declare, @Map, @Input y @Output no son estrictamente necesarias porque la información que proporcionan se puede extraer analizando el código. No se puede acceder al tamaño de los arrays dentro de los métodos paralelos. No hay tests, por lo que es difícil saber si una modificación está afectando a otras partes del código. No se distinguen los tipos de datos `float` y `double` en la clase Math de Java y las constantes de esta clase no se traducen.

Por otro lado hay *bugs*, por ejemplo, la memoria no se libera de correctamente debido a la no llamada del método `finalize`. Sólo se admite tener un constructor. La traducción de tipos de datos es ambigua en algunos casos y no nos podemos fiar de que estos datos existan aunque se procesen en varios lugares. Los `getters/setters` de los Bitmap son incorrectos. No se genera bien el código cuan-

do se sobrecargan los métodos. El código JNI está mezclado con el código de los métodos. El código OpenCL se inserta como un `string` dentro del código C/C++. Y los errores no se manejan correctamente en el compilador.

Tradux soluciona la mayoría de los problemas expuestos. Y, con su metodología de traducción, que se explica en el Apartado 3.6, se facilita que se puedan añadir modificaciones más fácilmente.

3.3. Integración con Fancier

La utilización de Fancier dentro de Tradux facilita la generación de código de varias maneras. La más evidente es que, al aportar una API común para Java, C/C++ y OpenCL se puede pasar de un lenguaje a otro fácilmente, permitiendo que la traducción sea inmediata. Además, Fancier aporta unos contenedores que resultan muy útiles para esto, ya que encapsulan los datos que nos interesan en interfaces equivalentes en cada uno de los lenguajes.

Por ejemplo, la clase `RGBAImage` de Java contiene los datos de los colores de los píxeles, la altura y la anchura de la imagen. En C existe un estructura `fcRGBAImage` equivalente que encapsula los mismos datos, y en OpenCL, que no se pueden crear estructuras de la misma forma que en C, se utiliza el tipo de dato `uchar4*` para almacenar los colores de los píxeles e `int` para el alto y el ancho de la imagen, los tres datos se pasan como parámetros a los *kernels*. Además de contenedores para imágenes, Fancier también incluye tipos de datos vectoriales, arrays, arrays de vectores y las clases `Math` y `RGBAColor`. Por tanto, utilizando Fancier se resuelven directamente algunos de los problemas de Paralldroid mencionados en el apartado anterior y se aumenta la cantidad de funcionalidades que se pueden utilizar. De esta forma, en Tradux se reduce la cantidad de código que se genera, ya que no hace falta generar el código para gestionar los objetos `Bitmap` específicos de Android ni el uso de arrays primitivos en la JNI.

Por otra parte, Fancier también resuelve el problema de la sincronización de memoria, evitando que hayan fugas. Además, gestiona automática y eficientemente la memoria, facilitando de nuevo la generación de código. Gracias a los métodos que encapsulan la inicialización y finalización de la memoria, los métodos de sincronización `syncToHost` y `syncToDevice`, y la gran variedad de excepciones C/C++ y OpenCL que se incluyen en Fancier el código que se genera con Tradux es mucho más sencillo y tiene menos problemas que el que se genera con Paralldroid.

3.4. Anotaciones

Al igual que Paralldroid, Tradux utiliza anotaciones para definir el paralelismo. Con Tradux se reduce la cantidad de estas, debido a que muchas se pueden inferir a partir de otros elementos que define el usuario o a que algunas se debían más a cuestiones internas que no deberían afectar a este. En la Tabla 3.1 se incluyen las anotaciones de Tradux.

Anotación	Aplicada a	Parámetro
@Kernel	Métodos	-
@Range	Métodos, parámetros de método	campo, dimensión

Tabla 3.1: Anotaciones de Tradux.

Estas dos anotaciones deben ir siempre acompañadas la una de la otra. @Kernel indica qué métodos van a ser paralelos y deben generarse como tal, y @Range define cómo se ejecuta el paralelismo; el número de dimensiones y el tamaño de cada una. En el Listado 3.1 podemos ver un ejemplo de su funcionamiento. Si comparamos con el Listado 2.1 observamos que la anotación @Kernel equivale a la anotación @Parallel de Paralldroid, y @Range a @NumThreads. El cambio de notación se debe a que con Tradux se usan términos más cercanos a la terminología de OpenCL en lugar de a OpenMP.

En el ejemplo del Listado 3.1, que suma una constante a todos los valores de un array, encontramos una llamada a la función `Parallel.GlobalID(0)` en la línea 8. Esto es una llamada a la API pública de Tradux, se trata de una clase llamada `Parallel` que permite acceder a métodos equivalentes a los que podrían llamarse desde un *kernel* de OpenCL durante su ejecución. Por ejemplo, con `Parallel.GlobalID(0)` se obtiene el índice global del hilo que se está ejecutando en la dimensión especificada, la 0 en este caso. La implementación de la API se puede encontrar en el Apéndice A.1 e incluye la clase `Parallel` y las anotaciones.

Por otra parte, el número de dimensiones y el tamaño de cada una está implícita en el tipo de dato al que acompaña la anotación @Range y el tamaño o el contenido de esta variable. En la Tabla 3.2 se encuentran los diferentes tipos de datos que pueden utilizarse con esta anotación.

3.5. Metodología de desarrollo

Para desarrollar un proyecto que utilice Tradux hay que tener en cuenta la estructura del proyecto, dependiendo de si se va a generar código para una aplicación Android o Linux el código se organiza de una manera u otra. Aunque existen opciones de línea de comando que permiten customizar la localización del código anotado y generado.

En el Listado 3.5 tenemos la estructura por defecto que se usa en un proyecto Android. Como puede verse en la figura hay cuatro directorios dentro de `src`; `assets` para el código OpenCL, `java` para el código Java implementado por el usuario y generado por Tradux, `jni` para el código C/C++ y `parall` para las clases anotadas que se quieren traducir. Cabe destacar que la estructura del directorio `parall` debe ser una réplica de la estructura del directorio `src`, pues las clases Java generadas se almacenan exactamente en el mismo punto donde estaba la clase anotada de origen. Por ejemplo, las clases Java generadas a partir de la clase anotada `Foo.java` en **`parall`**`/com/mypackage/` estarán en **`src`**`/com/mypackage/`.

```
.
|-- app
    |
    |-- CMakeLists.txt
    |-- cmake
    |   |-- libfancier.so, libOpenCL.so, ...
    |-- libs
    |   |-- fancier-android-1.0.jar, paralldroid_api.jar, ...
    |-- src
        |-- main
            |-- assets
            |   |-- fc_image.cl, fc_math.cl, FooFOCL.cl, ...
            |-- java/com/mypackage
            |   |-- Foo.java, FooFJ.java, FooFN.java, FooFOCL.java
            |   |-- MyOtherClass.java, MainActivity.java, ...
            |-- jni
            |   |-- include
            |   |   |-- CL/, fancier/, ...
            |   |   |-- FooFN.h, FooFOCL.h, ...
            |   |-- src
            |       |-- FooFN.c, FooFOCL.c, ...
            |-- parall/com/mypackage
                |-- Foo.java, ...
```

Listado 3.5: Estructura de un proyecto Android en Tradux.

Tipo de dato		Dimensiones	Tamaño/dimensión
Primitivo	int	1	var
Vector	Int2, Float2, Double2, Short2, Byte2, Long2	2	var.x, var.y
	Int3, Float3, Double3, Short3, Byte3, Long3	3	var.x, var.y, var.z
	Int4, Float4, Double4, Short4, Byte4, Long4	4	var.x, var.y, var.z, var.w
	Int8, Float8, Double8, Short8, Byte8, Long8	8	var.x, var.y, var.z, var.w var.s[0], var.s[1], var.s[2], var.s[3]
Array	IntArray, FloatArray, DoubleArray, ShortArray, ByteArray, LongArray	1	var.length()
Image	RGBAImage	2	var.getWidth(), var.getHeight()

Tabla 3.2: Tipos de datos que acompañan a @Range, `var` representa una variable instanciada de cada tipo.

Con cada ejecución del traductor se genera todo el código necesario para todos los *backends* disponibles: Fancier Java, Fancier Native y Fancier OpenCL, además de una clase que sirve de interfaz que es con la que trabaja el usuario. De esta forma las diferentes implementaciones son intercambiables y el usuario no nota diferencias al trabajar con una u otra. En la Figura 3.1 se encuentran representados los diferentes *backends* y todos los ficheros que se generan para una clase anotada `Foo.java`. Si se quisiesen hacer modificaciones al código generado bastaría con cambiarle el nombre al fichero para que este no se sobreescriba con nuevas ejecuciones de Tradux.

La interfaz que se genera se inspira en los patrones de diseño Plantilla y Estrategia. Esta interfaz permite abstraer la estructura de la implementación de las clases, tener una referencia del código que hace falta generar para nuevos *backends* e intercambiar las subclases generadas. Cosa que se proponía en Paralldroid pero no llegaba a conseguirse. En el Listado 3.6 podemos ver cómo se utilizaría una clase generada con la interfaz.

3.6. Metodología de transformación

Respecto a la metodología de traducción que utiliza Tradux para la generación del código destaca la separación que hace entre el análisis y la transformación del código, que se hace en diferentes etapas, Figura 3.2. De esta forma la traducción es iterativa y favorece la reutilización del código. El funcionamiento básico de esta metodología es que en cada fase de traducción, que se lleva a cabo por una clase

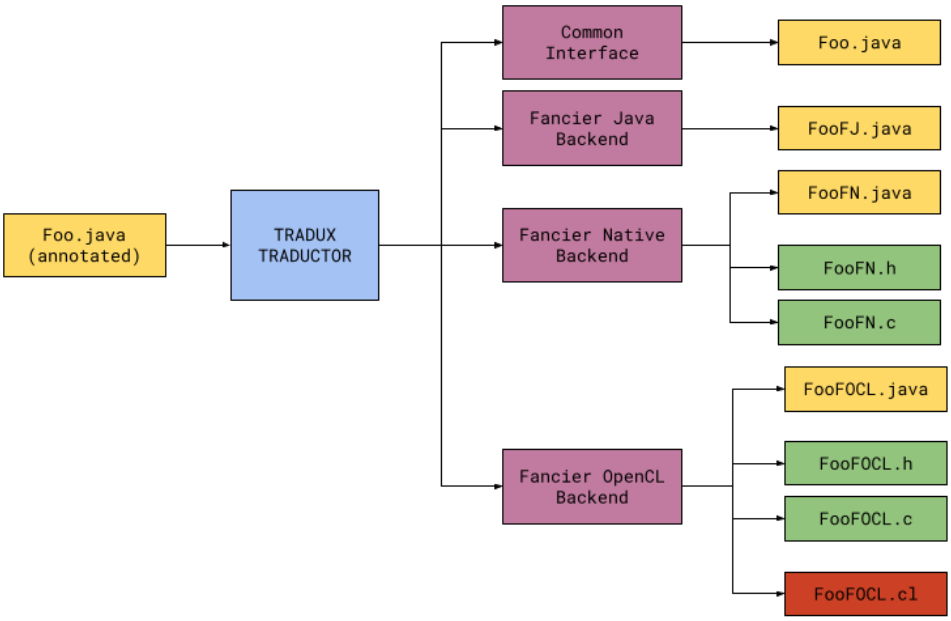


Figura 3.1: Ficheros generados con Tradux.

independiente, se genera un AST intermedio parcialmente traducido que se refina en la siguiente etapa. Así, todos los traductores reciben y producen código inválido hasta que se llega a la última etapa. La principal ventaja de modularizar el código de este modo es que se facilitan futuros cambios y extensiones de la librería.

En este apartado se desglosan los principales pasos que se realizan dentro de Tradux cada vez que ejecuta. El código se encuentra en un respositorio de GitHub [7]. Tradux utiliza el parseador de OpenJDK para obtener los AST de las clases anotadas por el usuario y una vez obtenidos se ejecuta el código para generar cada *backend*. Aunque la interfaz conceptualmente no es un *backend* en la implementación se trata como tal, ya que tiene sus propias fases de refinamiento. `Backend` es una clase abstracta que define dos métodos que debe tener la implementación de cada *backend* e incluye otros datos que comparten las instancias de cada uno de ellos, los métodos más importantes son `instance`, que inicializa todos los traductores y procesadores de cada *backend*, y `translate`, que ejecuta los traductores y procesadores inicializados. Los procesadores son las clases encargadas de modificar iterativamente los ASTs, cada uno se encarga de añadir una funcionalidad o característica específica al AST. El nivel de especificación de estos lleva, en algunos casos, a necesitar de clases subprocesadores, como por ejemplo con la traducción de los tipos de datos.

Es importante señalar la diferencia entre **traductor** y **procesador**. El primero crea, a partir de los nodos de los ASTs entrada, otros nodos Java, C u OpenCL, según corresponda. Mientras que el segundo hace referencia al proceso de refinamiento de los nodos. Dentro de cada *backend* también se hace uso de clases constructoras y analizadores, además de otras clases que facilitan la traducción de nombres y tipos de datos de un lenguaje a otro. Pero por simplicidad nos centramos en los procesos de análisis, traducción y refinación.

```
1  public class Main {
2    public static void main(String[] args) {
3      ...
4      FloatArray arr = new FloatArray(new float[]{ 0, 1, 2, 3, 4,
5                                                   5, 6, 7, 8, 9 });
6      float kConstant = 1;
7
8      /** Fancier Java Performance */
9      FooFJ strategy = new FooFJ();
10     /** Fancier Native */
11     FooFN strategy = new FooFN();
12     /** Fancier OpenCL */
13     FooFOCL strategy = new FooFOCL();
14
15     /** Common Interface */
16     Foo context = new Foo(strategy);   // Chosen strategy
17     context.run(arr, kConstant);
18     ...
19   }
20 }
```

Listado 3.6: Main de ejemplo simple en Tradux.

3.6.1. Análisis

Las fases de análisis son importantes debido a que aportan información sobre la estructura semántica del AST de entrada, y esta información se puede usar luego en varios lugares. De este modo, al descomponer el trabajo en varias clases específicas se puede reutilizar el código y simplificar los traductores y procesadores. La mayoría de clases analizadoras son comunes para todos los *backends* y se encargan de analizar las anotaciones, el uso de variables globales y locales, la inicialización de los atributos, etc. Pero hay también clases específicas para cada *backend*, por ejemplo, para el código Java hay dos analizadores; uno para las variables que se predeclaran en los *kernels* y otro para la creación de arrays estáticos dentro de estos. En C también se incluye un análisis para distinguir entre campos de instancia y estáticos.

3.6.2. Traducción

Un problema que tiene Paralldroid es que utiliza las clases de OpenJDK para trabajar con el AST de C, y en consecuencia, también con el de OpenCL. En Tradux se han creado clases, basadas en las de Java, para poder crear un AST C, con las funcionalidades que se necesitan. Para esto se utilizan las clases `CTree`, `CTreeElements`, `CTreeFactory` y `CTreeScanner` en C. Y las clases `OCLTree`, `OCLTreeElements`, `OCLTreeFactory` y `OCLTreeScanner` en OpenCL.

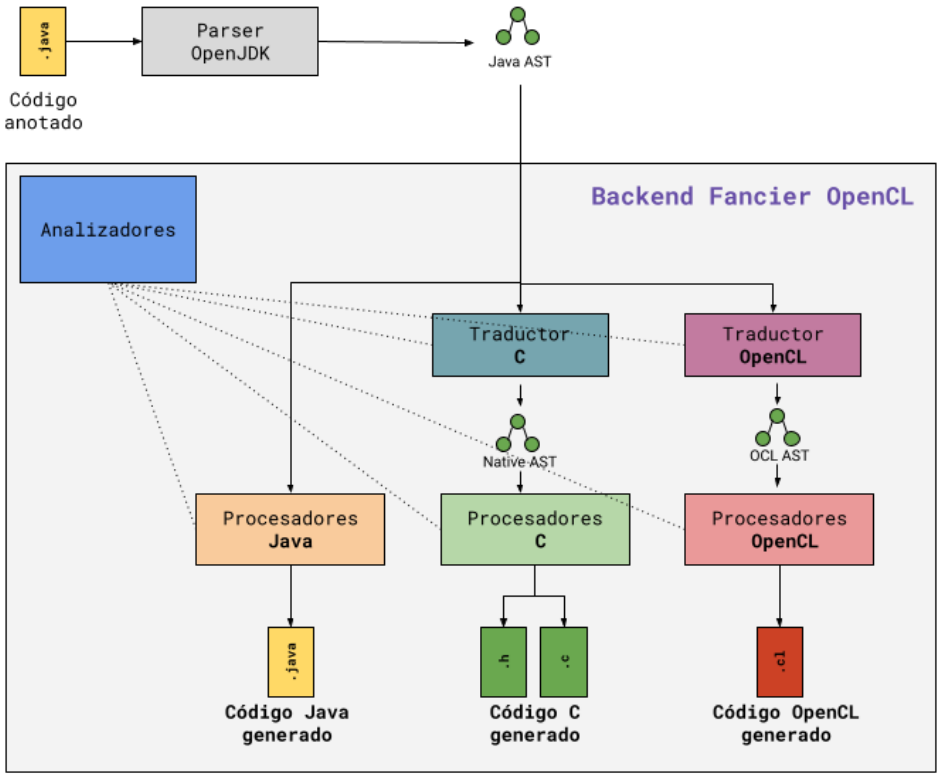


Figura 3.2: Proceso de traducción general del *backend* Fancier OpenCL.

Estas clases son usadas por las clases que traducen los nodos Java a nodos C; `CTranslator` y sus subclases `CBasicTranslator`, `CSubTranslator`, `CForeachSubTranslator`, `CObjectDeclSubTranslator`, `CPrimitiveSubTranslator` y `CTypeSubTranslator`. Y las clases que traducen los nodos Java a nodos OpenCL, `OCLTranslator` y sus subclases. OpenCL es un sublenguaje de C, por lo que las clases OpenCL heredan la mayor parte de su funcionalidad de las clases de C.

3.6.3. Refinación

El proceso de refinación de cada AST es diferente, como se explicó al principio de este apartado, cada clase procesadora es independiente y aporta algo diferente al AST de salida. En la Figura 3.3 se encuentran los procesadores utilizados en el *backend* Fancier OpenCL. Como vemos, cada fichero generado, es decir, cada lenguaje, tiene sus propios procesadores. Y dependiendo del *backend* para el que se está generando el código cada lenguaje se utilizan unos u otros.

3.7. Características y restricciones

Las características y restricciones de Tradux se incluyen en este apartado.

Características:

- El código de los *kernels* se tiene que "escribir en Fancier". Es decir, sólo se pueden utilizar tipos de datos primitivos de Java, llamadas a la API de Tradux

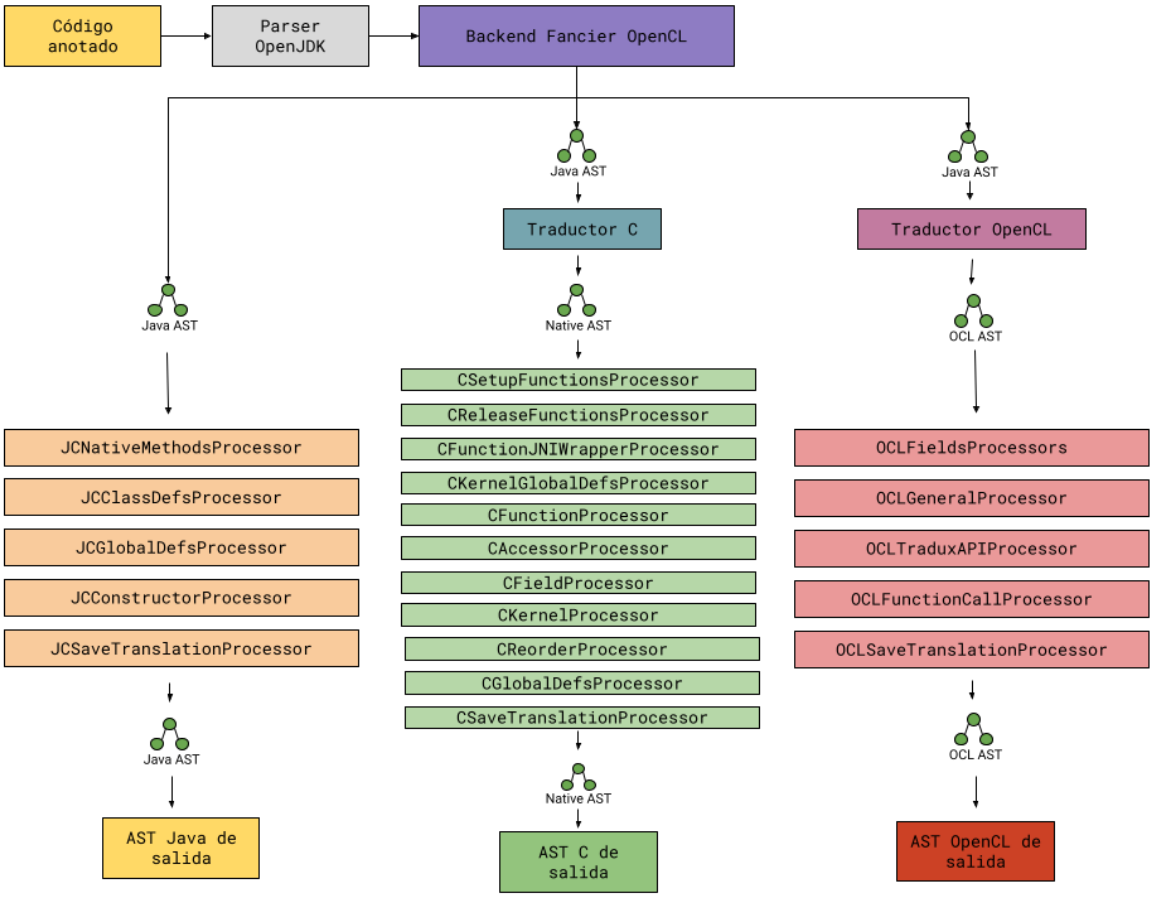


Figura 3.3: Traductores y procesadores del *backend* Fancier OpenCL.

y objetos y tipos de datos Fancier. Por ejemplo, se pueden utilizar los arrays de Fancier pero no los de Java.

- Como excepción se pueden utilizar arrays estáticos dentro de los *kernels*.
- Se pueden llamar a métodos de la misma clase desde el *kernel*.
- Los vectores de Fancier permiten definir ejecuciones paralelas multidimensionales.
- Solo se generan los `getters` y los `setters` que el usuario implemente en la clase anotada.
- El código se genera en el paquete equivalente de `src` que el de `parall` en que se encuentra la clase anotada.
- Los únicos métodos públicos que se generan en lenguajes diferentes a Java son los anotados con @Kernel.
- Los métodos privados se generan en todos los lenguajes necesarios para que estos sean accesibles por Java, C/C++ y OpenCL según corresponda a cada *backend*.

Restricciones:

- No se puede usar cualquier objeto dentro de las clases anotadas. Por ejemplo, no se soporta el uso de la clase String de Java.
- El tamaño de los arrays estáticos tiene que ser un literal entero.
- @Kernel solo puede anotar método públicos.
- @Range no puede anotar variables de tipo arrays de vectores.
- Se tienen las mismas restricciones que tiene OpenCL como lenguaje. Por tanto, no se pueden crear contenedores dentro de los *kernels*, estos se tienen que pasar como parámetros.
- No se soporta la recursividad.
- Los métodos públicos sólo se generan en Java, excepto los anotados con @Kernel.

3.8. Testing

Por último, debido a la dificultad de depurar el código generado, por su extensión, surgió la necesidad de diseñar una metodología que permitiese comprobar que el código que generado es el deseado. De esta manera se consigue identificar rápidamente cuándo una modificación está afectando a una parte del código a la que no debería.

El funcionamiento de la metodología es simple, en ficheros de texto se escribe el código de salida que se espera obtener de cada clase anotada. Como si se tratase de una ejecución normal se leen las clases anotadas del directorio de las pruebas y se obtienen los ASTs. Estos se pasan al proceso normal de traducción y generación de código, aunque se le hace alguna modificación al código generado para facilitar la comparación de los ficheros y evitar problemas con los nombres de los paquetes. Una vez que generado el fichero se parsea y se compara con el fichero de referencia. Cuando los ficheros no coinciden la aplicación lanza un error y dice la línea en la que no coincide el código generado con el esperado. Si todo va bien se muestra un mensaje diciendo que las pruebas son correctas.

Capítulo 4

Experiencia computacional

En este capítulo se incluyen los resultados computacionales obtenidos con Tradux. Se muestran las gráficas de las pruebas de rendimiento realizadas, se comenta la plataforma hardware utilizada y la dificultad de medir tiempos en este tipo de plataformas, y además se incluyen los diferentes algoritmos que se ejecutaron.

4.1. Plataforma hardware

Las pruebas se realizaron en un dispositivo **Huawei P8 Lite 2017**, en adelante **P8L**. Este dispositivo tiene las especificaciones que se muestran en la Tabla 4.1. De este dispositivo cabe destacar la arquitectura de su CPU, que es Arm big.LITTLE, por lo que cuatro de sus núcleos son más lentos y consumen menos energía (LITTLE) y los otros cuatro son más rápidos pero consumen más batería (big).

	Huawei P8 Lite
SSOO	Android 8.0
SoC	HiSilicon Kirin 655
CPU	Arm Cortex-A53 (4-core @ 2.1 GHz) + Arm Cortex-A53 (4-core @ 1.7 GHz)
CPU	Arm Mali-T830 MP2 (2 CU @ 900 MHz)
RAM	3 GB LPDDR3 @ 933 MHz
Año	2017

Tabla 4.1: Especificaciones Huawei P8 Lite 2017.

4.2. Fiabilidad de las pruebas

En los dispositivos móviles es difícil realizar mediciones, ya que obtener resultados representativos y precisos es complicado. Esto se debe a que las características de los SoC son cambiantes. Por ejemplo, el nivel de batería influye en cómo el sistema operativo controla las aplicaciones que se están ejecutando. Es decir, si una aplicación está consumiendo mucha batería y está utilizando los núcleos más rá-

pidos de la CPU el sistema puede elegir cambiar automáticamente y empezar a ejecutar la aplicación con los núcleos más lentos. De esta forma, dos ejecuciones del mismo programa pueden dar lugar a resultados muy diferentes. Por este motivo se utiliza Rancid [2], una librería que se encarga de estabilizar el sistema, permitiendo que el dispositivo móvil tenga las mismas características cada vez que se ejecuta el programa. Las principales variables que se pueden controlar con Rancid son la frecuencia del procesador y la temperatura.

4.3. Procesamiento de imágenes

Para analizar el rendimiento de Tradux se han implementado varios filtros de imágenes en diferentes versiones. Las versiones son las siguientes:

- **Bitmap Java**: esta implementación se hizo de forma manual como referencia a lo que sería una implementación habitual en Android utilizando el tipo de dato `Bitmap`.
- **Fancier Java**: es una implementación manual que utiliza la API Fancier. Se trata de una versión simple en la que se crean muchos objetos pequeños.
- **Fancier Java Performance**: esta es la versión Java generada con Tradux, donde se utiliza Fancier pero se inicializan las variables y se obtienen los *buffers* de los contenedores fuera de los bucles.
- **Bitmap Native**: esta versión es la que se toma de referencia para la comparación de los resultados. Se trata de una implementación manual en C que utiliza objetos `Bitmap`.
- **Fancier Native**: esta es la versión C generada con Tradux, que utiliza la API de Fancier.
- **Fancier OpenCL**: esta es la versión OpenCL generada con Tradux. El resultado es un código paralelo que se puede ejecutar en la GPU del dispositivo móvil.

Los filtros de imágenes utilizados se incluyen a continuación:

- **Grayscale (GS)**: filtro que convierte una imagen a color a escala de grises.
- **Blur (GB)**: es un filtro de suavizado basado en la función Gaussiana.
- **Convolve 3x3 (C3)**: filtro lineal en que cada píxel es una media ponderada de sus 8 píxeles más cercanos.
- **Convolve 5x5 (C5)**: filtro lineal en que cada píxel es una media ponderada de sus 24 píxeles más cercanos.
- **Bilateral (BL)**: es un filtro que suaviza los píxeles en función de los píxeles vecinos y su distancia al centro.

- **Median (ME)**: este filtro aplica la intensidad media de los de los píxeles vecinos a cada píxel, en esta implementación solo se utiliza el canal rojo.
- **Contrast (CO)**: filtro que mejora el contraste de una imagen.
- **Fisheye (FE)**: este filtro aplica la distorsión de una lente a la imagen de entrada.
- **Levels (LV)**: filtro que modifica el brillo, el color y el contraste de la imagen.
- **Posterize (PO)**: filtro en el que a cada píxel se le asigna el color más cercano de entre unos colores preseleccionados.

Cada combinación entre filtro e implementación se ejecutó 10 veces con ocho resoluciones de imágenes diferentes, que son las siguientes: **VGA** (640 x 480), **XGA** (1024 x 768), **HD1** (1280 x 720), **HD2** (1366 x 768), **HD+** (1600 x 900), **FHD** (1920 x 1080), **QHD** (2560 x 1440) y **UHD** (3840 x 2160). Cada ejecución se lanzó 30 segundos después de que acabase la anterior para que el dispositivo reposase y volviese a sus características iniciales.

4.4. Resultados

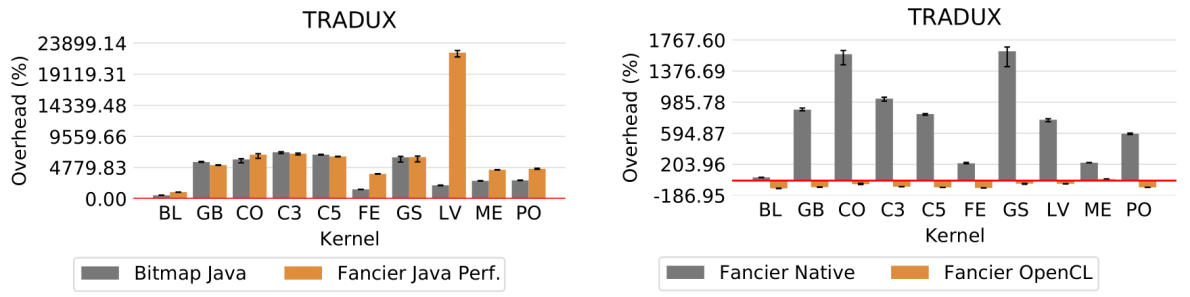
Tras la ejecución de las pruebas y la generación de los gráficos se procedió a la interpretación de los resultados. De esta forma, en este apartado se explican los resultados generales que se obtuvieron para todos los filtros y el caso concreto del filtro **Posterize**.

En primer lugar se comentan los resultados generales, que se pueden ver en la Figura 4.1. Debido a la diferencia de escala se han separado los gráficos en dos imágenes, a la izquierda se encuentran los resultados obtenidos para las implementaciones Java y a la derecha los de las nativas. Los primeros siempre dan peores resultados que la implementación de referencia `Bitmap Native`, cosa esperable debido a que C/C++ es más eficiente. Aún así, cabe destacar que las implementaciones `Bitmap Java` y `Fancier Java Performance` dan resultados similares para casi todos los filtros. Aunque los resultados presentados en Fancier [3] demostraban que `Fancier Java Performance` era significativamente mejor que `Bitmap Java`.

En la Figura 4.1(a) destaca especialmente el resultado obtenido con el filtro **Levels**. Este puede explicarse debido a que hay una llamada a una función desde el *kernel*, que en la implementación manual se hacía fuera de los bucles. Pero, al usar Tradux, debido a sus restricciones, se tiene que hacer dentro de estos. De esta forma se evidencia que en algunos casos para obtener mejoras de rendimiento no todo se puede automatizar y hará falta una etapa de tuneado del código por parte del usuario.

Por otra parte, los resultados referentes a las implementaciones nativas, Figura 4.1(b), obtienen los resultados esperados. Por un lado la implementación `Fancier Native` obtiene mejores resultados que las implementaciones Java, pero no mejores que la de referencia. No obstante los resultados de esta implementación

todavía tienen margen de mejora si se trabaja con los punteros que apuntan a los píxeles de las imágenes, en vez de utilizar los métodos de acceso a los contenedores de Fancier. Por último, los resultados de OpenCL sí que obtienen mejoras de rendimiento, ya que se ejecutan en un procesador especializado, la GPU. Se obtiene alrededor de un 90 % de mejora en el tiempo de ejecución comparado con la implementación `Native Bitmap`.



(a) Implementaciones Java.

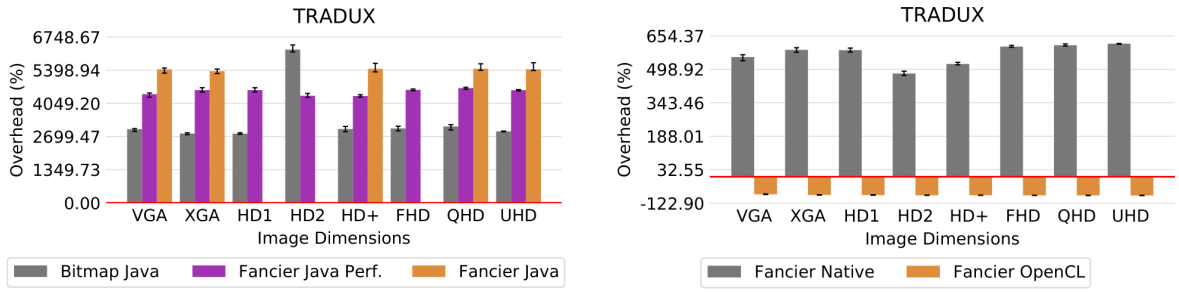
(b) Implementaciones nativas.

Figura 4.1: Resumen de resultados para todos los filtros de imágenes.

En segundo lugar se describen los resultados obtenidos con el filtro **Posterize** para cada resolución de imagen. El código generado en Tradux para este filtro se encuentra en el Apéndice B. En la Figura 4.2 se encuentran las gráficas correspondientes a los resultados de este filtro. En la Figura 4.2(a) tenemos los resultados de las implementaciones Java, en este caso se incluyen también los resultados de la implementación `Fancier Java`, ya que en la mayoría de las resoluciones de imágenes se obtuvieron datos. En la figura anterior no aparecía esta implementación debido a que muchos datos no se pudieron calcular porque se trata de una versión muy lenta que en muchos filtros llegó a la condición de parada del algoritmo antes de que este acabase de ejecutarse.

En relación a los resultados de estas implementaciones vemos que la versión `Bitmap Java` es la mejor, seguida de la de `Fancier Java Performance` y la de `Fancier Java` para la mayor parte de las resoluciones. Esto indica que independientemente del tamaño del problema la escalabilidad se mantiene. Por otra parte, en la Figura 4.2(b) tenemos los resultados de las implementaciones nativas, donde la versión `Fancier Native` obtuvo peores resultados que la versión `Bitmap Native` de referencia. Y la versión paralela de OpenCL sí que obtuvo mejores resultados.

Por tanto, Tradux consigue su objetivo, genera código más eficiente en dispositivos móviles cuando utiliza el *backend* Fancier OpenCL. Sin embargo, los otros dos *backends* no logran mejorar el rendimiento de las implementaciones equivalentes de su mismo lenguaje. No obstante, todavía hay margen de mejora en el caso de la versión Fancier Native.



(a) Implementaciones Java.

(b) Implementaciones nativas.

Figura 4.2: Resultados del filtro Posterize.

Capítulo 5

Conclusiones y líneas futuras

En este capítulo se exponen las conclusiones a las que se ha llegado y las posibles líneas de trabajo futuro que podrían seguirse.

5.1. Conclusiones

Se ha presentado una aplicación multiplataforma, Tradux, que puede utilizarse tanto en Android como en Linux, capaz de generar código paralelo eficiente de forma automática a partir de código Java anotado. Desde código aparentemente secuencial se consigue generar código equivalente en Java, C/C++ y OpenCL.

Partiendo de Paralldroid y utilizando Fancier se ha conseguido crear una librería que soluciona la mayor parte de los problemas de la primera, a la vez que, al usar la segunda, aumenta el número de funcionalidades. También se utiliza una nueva metodología de traducción basada en el análisis y el refinamiento progresivo del código generado, lo que facilita la futura extensibilidad de la aplicación. Se han conseguido desarrollar tres *backends* diferentes, Fancier Java, Fancier Native y Fancier OpenCL; para ejecutar código en Java, C/C++ y OpenCL respectivamente. Además se ha desarrollado una metodología de testing que facilita la depuración.

Finalmente, las pruebas de rendimiento realizadas a filtros de imágenes demuestran que, al utilizar el código generado en OpenCL, Tradux mejora el rendimiento que se obtiene en implementaciones Java manuales. Como se esperaba, el código paralelo obtiene mejores resultados que los otros código secuenciales generados e implementados.

5.2. Líneas futuras

A pesar de haber conseguido lo que se había propuesto inicialmente con Tradux todavía quedan mejoras con las que se podría ampliar este trabajo en el futuro. A nivel conceptual hay varias líneas de trabajo que podrían resultar especialmente interesantes, son las siguientes:

- Realizar las pruebas de rendimiento en otro dispositivo para poder comparar los resultados, principalmente debido a que el P8L en el que se realizaron las

pruebas tiene varios años ya. Además, sería interesante probar un dispositivo con una versión de más nueva de Android donde Renderscript ya no esté soportado.

- Implementar una aplicación de prueba más compleja donde se resuelva un problema como el Filtro de Partículas, una aplicación tipo SLAM o algún proyecto que resuelva problemas dirigidos al aprendizaje automático.
- Desarrollar una aplicación que compare Tradux con otra librería con su mismo objetivo, Aparapi, para analizar su validez dentro del sector.
- Soportar la generación de código paralelo desde el código nativo, por ejemplo utilizando OpenMP en el *backend* Fancier Native.
- Probar la portabilidad en otros aceleradores como DSPs o FPGAs.

Por otra parte, modificaciones en el código que mejoren las características de la librería también se contemplan de cara a trabajo futuro, por ejemplo:

- Permitir la utilización de clases y objetos externos que provengan de otras clases anotadas.
- En el *backend* Fancier Native utilizar los punteros que apuntan al contenido de los objetos de la API de Fancier, en lugar de utilizar los métodos de acceso de los contenedores.
- Soportar la concurrencia.

Capítulo 6

Summary and conclusions

This chapter includes the main conclusions of the project and the future work that can be done.

6.1. Conclusions

We have presented Tradux, a multiplatform application which can be used both in Android and Linux. It can generate efficient parallel code automatically from Java-annotated code. Java, C/C++ and OpenCL codes are generated from apparently sequential code.

Based on Paralldroid, Tradux solves most of its problems, and using Fancier, it increases its functionalities. A new translation methodology is also used. This methodology is based on the analysis of the original code and the iterative refinement of the generated code, easing the extensibility of the library. Three backends have been developed; Fancier Java for Java code, Fancier Native for C/C++ code and Fancier OpenCL for OpenCL code. Moreover, a new testing methodology has been implemented too.

Finally, the performance tests, made to image filters, show that the OpenCL-generated codes achieve performance improvements. As expected, the parallel OpenCL code is more efficient than the sequential codes generated by the other backends or the manual implementations of the other languages.

6.2. Future work

Despite achieving the initial goals of Tradux, we still have some improvements for the future. On a conceptual level, different approaches can be especially interesting, such as the following:

- Execute the performance tests on another device to compare the results we have. It would also be interesting to test a device with a newer version of Android where Renderscript is not supported anymore.
- Implement a complex test application for resolving problems such as the

Particle Filter, the SLAM problem or tasks related to Machine Learning.

- Develop an application to compare Tradux with Aparapi, because comparing it with another application in the sector would let us analyze its validity.
- Support parallel code generation from the native code, for example, using OpenMP on the Fancier Native backend.
- Test the portability on other accelerators such as FPGAs and DSPs.

Additionally, programmatic achievements to improve the library's characteristics are also kept in mind, for example:

- Allow the use of objects that come from other annotated classes.
- Use pointers directly instead of the Fancier accessor methods for containers inside the Fancier Native backend.
- Support concurrency.

Capítulo 7

Presupuesto

En este capítulo se presenta el presupuesto que se ha estimado que conlleva el desarrollo del proyecto. Se incluyen los gastos de recursos humanos y el hardware utilizado.

Los recursos humanos se limitan a una desarrolladora, que es quien ha implementado la librería Tradux. Por otra parte, el hardware que ha sido necesario es el ordenador utilizado para el desarrollo, una pantalla auxiliar y el móvil usado para la realización de las pruebas. En la Tabla 7.1 se incluye el coste total de los nueve meses de duración de la asignatura Trabajo Fin de Máster.

Tipo	Descripción	Coste	Coste total
Recursos humanos	1 Desarrolladora	1500€/mes	13500€
Equipo	Lenovo ThinkPad L13 Yoga	1000€	1000€
Equipo	Pantalla MSI Optix G24	130€	130€
Pruebas	Huawei P8 Lite 2017	235€	235€
Total			14865€

Tabla 7.1: Estimación del presupuesto del proyecto.

Apéndice A

API de Tradux

A.1. Clase Parallel

```
1  package es.ull.hpc.pcg.paralldroid.api;
2
3
4  /**
5   * Use OpenCL interface for writing kernels
6   * Includes Work-Item functions specified in OpenCL documentation
7   * https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf
8   */
9  public class Parallel {
10
11     /**
12      * NDRange index space, what it means, the number of dimensions
13      * that are in use
14      * @return the number of dimensions
15      */
16     public static int WorkDim() {
17         return 1;
18     }
19
20     /**
21      * @param dimId dimension ID
22      * @return number of global work-items of the specified dimension
23      */
24     public static long GlobalSize(int dimId) {
25         return 1L;
26     }
27
28     /**
29      * Global ID used to uniquely identify a work-item when
30      * executing a kernel
31      * @param dimId dimension ID
```

```
32      * @return the unique global work-item ID of the specified
33      * dimension
34      */
35     public static long GlobalID(int dimId) {
36         return 0L;
37     }
38 
39     /**
40      * @param dimId dimension ID
41      * @return number of local work-items of the specified dimension
42      */
43     public static long LocalSize(int dimId) {
44         return 1L;
45     }
46 
47     /**
48      * ID used inside a work-group
49      * A composition of a group and a local ID are needed to
50      * identify an item
51      * @param dimId dimension ID
52      * @return the unique local work-item ID of the specified
53      * dimension
54      */
55     public static long LocalID(int dimId) {
56         return 0L;
57     }
58 
59     /**
60      * @param dimId dimension ID
61      * @return the number of work-groups that will be executed by a
62      * kernel for the specified dimension
63      */
64     public static long NumGroups(int dimId) {
65         return 1L;
66     }
67 
68     /**
69      * ID assigned to a work-group
70      * A composition of a group and a local ID are needed to identify
71      * an item
72      * @param dimId dimension ID
73      * @return groupID
74      */
75     public static long GroupID(int dimId) {
76         return 0L;
77     }
78 
```

```
79      /**
80       * @param dimId dimension ID
81       * @return the offset of the specified dimension
82       */
83      public static long GlobalOffset(int dimId) {
84          return 0L;
85      }
86
87  }
```

A.2. Anotación @Kernel

```
1  package es.ull.hpc.pcg.paralldroid.api.annotations;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8
9  /**
10  * Tell the Paralldroid compiler that methods annotated with it
11  * contains a kernel, this kernel will be executed using a
12  * predefined number of threads defined with @Range annotation
13  */
14 @Target(ElementType.METHOD)
15 @Retention(RetentionPolicy.CLASS)
16 public @interface Kernel {
17 }
```

A.3. Anotación @Range

```java
package es.ull.hpc.pcg.paralldroid.api.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;


/**
 *  This annotation defines how many threads are launched
 *  concurrently, or, if the used backend is not parallel
 *  how many loops have to be generated
 *
 *  It can be applied to a method or to a method parameter, provided
 *  that this method has been annotated with @Kernel too
 */
@Target({ ElementType.METHOD, ElementType.PARAMETER })
@Retention(RetentionPolicy.CLASS)
public @interface Range {

    /**
     * @return which field of the class set the number of threads
     */
    String field() default "";

    /**
     * @return how many dimensions the program is working with
     */
    int dims() default 0;

}
```

Apéndice B

Código generado para el filtro Posterize

En este apéndice se incluye el código que se ha generado para uno de los filtros implementados, el filtro **Posterize** para plataformas Android. En primer lugar se muestra el código anotado por el usuario y luego se incluye el código generado para cada uno de los *backends* de Tradux; Fancier Java, Fancier Native y Fancier OpenCL.

B.1. Código Java anotado

```
1  public class Posterize {
2
3    private static Float3 GS_WEIGHTS = new Float3(0.229f, 0.587f, 0.114f);
4
5    private static Float2Array INTENSITIES = new Float2Array(new float[]{
6      0.0f, 0.2f, 0.2f, 0.4f, 0.4f, 0.6f, 0.6f, 0.8f, 0.8f, 1.0f,
7    });
8
9    private static Byte4Array COLORS = new Byte4Array(new byte[]{
10     RGBAColor.RED().x, RGBAColor.RED().y, ..., RGBAColor.CYAN().w,
11   });
12
13   private int width;
14   private int height;
15
16   public Posterize(int width, int height) {
17     this.width = width;
18     this.height = height;
19   }
20
21   @Kernel
22   public void run(@Range RGBAImage input, RGBAImage output) {
23     int x = (int) Parallel.GlobalID(0);
24     int y = (int) Parallel.GlobalID(1);
25
26     for (int stage = 0; stage < INTENSITIES.length(); ++stage) {
27       Float2 intensity = INTENSITIES.get(stage);
28       Byte4 color = COLORS.get(stage);
29       Byte4 pixel = input.get(x, y);
30
```

```
31        float pixel_intensity = Float3.dot(
32            Float3.div(pixel.asByte3().convertFloat3(), 255f), GS_WEIGHTS);
33
34        if( (pixel_intensity <= intensity.y) && (pixel_intensity >= intensity.x))
35          output.set(x, y, color);
36      }
37    }
38  }
```

B.2. Interfaz generada

```
1  public class Posterize {
2    private static Posterize ref;
3    protected AssetManager assets;
4    protected static Float3 GS_WEIGHTS = new Float3(0.229F, 0.587F, 0.114F);
5    protected static Float2Array INTENSITIES =
6              new Float2Array(new float[]{0.0F, 0.2F, ..., 1.0F});
7    protected static Byte4Array COLORS =
8              new Byte4Array(new byte[]{RGBAColor.RED().x, ..., RGBAColor.CYAN().w});
9    protected int width;
10   protected int height;
11
12   protected Posterize(String fancierInitPath, AssetManager assets) {
13     this.assets = assets;
14     Fancier.init(fancierInitPath);
15   }
16
17   public Posterize(Posterize backend) {
18     ref = backend;
19   }
20
21   protected void finalize() {
22     Fancier.release();
23   }
24
25   @Kernel()
26   public void run(@Range() RGBAImage input, RGBAImage output) {
27     ref.run(input, output);
28   }
29 }
```

B.3. Código Fancier Java generado

B.3.1. Código Java

```
1  public class PosterizeFJ extends Posterize {
2
3    public PosterizeFJ(String fancierInitPath, AssetManager assets, int width,
4                       int height) {
5      super(fancierInitPath, assets);
6      this.width = width;
7      this.height = height;
8    }
9
10   @Kernel()
11   public void run(@Range() RGBAImage input, RGBAImage output) {
12     ByteBuffer __paralldroidBuffer_param_input = input.getBuffer();
13     ByteBuffer __paralldroidBuffer_param_output = output.getBuffer();
14     ByteBuffer __paralldroidBuffer_field_INTENSITIES = INTENSITIES.getBuffer();
15     ByteBuffer __paralldroidBuffer_field_COLORS = COLORS.getBuffer();
16     Float2 intensity = new Float2();
17     Byte4 color = new Byte4();
18     Byte4 pixel = new Byte4();
19     Float3 __paralldroid_tmp0 = new Float3();
20     Byte3 __paralldroid_tmp1 = new Byte3();
21     Float3 __paralldroid_tmp2 = new Float3();
22     int __paralldroid_dims = 2;
23     int[] __paralldroid_index = new int[__paralldroid_dims];
24     int[] __paralldroid_sz = new int[]{input.getWidth(), input.getHeight()};
25     for (__paralldroid_index[0] = 0;
26          __paralldroid_index[0] < __paralldroid_sz[0]; ++__paralldroid_index[0]) {
27       for (__paralldroid_index[1] = 0;
28            __paralldroid_index[1] < __paralldroid_sz[1]; ++__paralldroid_index[1]) {
29         int x = (int)__paralldroid_index[0];
30         int y = (int)__paralldroid_index[1];
31         for (int stage = 0; stage < INTENSITIES.length(); ++stage) {
32           Float2Array
33                .getBuffer(__paralldroidBuffer_field_INTENSITIES, stage, intensity);
34           Byte4Array.getBuffer(__paralldroidBuffer_field_COLORS, stage, color);
35           input.getBuffer(__paralldroidBuffer_param_input, x, y, pixel);
36           pixel.asByte3(__paralldroid_tmp1);
37           __paralldroid_tmp1.convertFloat3(__paralldroid_tmp2);
38           Float3.div(__paralldroid_tmp2, 255.0F, __paralldroid_tmp0);
39           float pixel_intensity = Float3.dot(__paralldroid_tmp0, GS_WEIGHTS);
40           if ((pixel_intensity <= intensity.y) &&
41              (pixel_intensity >= intensity.x)) {
42             output.setBuffer(__paralldroidBuffer_param_output, x, y, color);
43           }
44         }
45       }
46     }
47   }
48 }
```

B.4. Código Fancier Native generado

B.4.1. Código Java

```
1  public class PosterizeFN extends Posterize {
2    static {
3      System.loadLibrary("PosterizeFN-lib");
4    }
5    private long nativeInstancePtr = 0L;
6    private static int instanceCount = 0;
7
8    public PosterizeFN(String fancierInitPath, AssetManager assets, int width,
9                       int height) {
10     super(fancierInitPath, assets);
11     this.width = width;
12     this.height = height;
13     if (instanceCount <= 0) {
14       instanceCount = 0;
15       setupClass(GS_WEIGHTS, INTENSITIES, COLORS);
16     }
17     ++instanceCount;
18     setupInstance(width, height);
19   }
20
21   @Kernel()
22   public native void run(@Range() RGBAImage input, RGBAImage output);
23
24   protected void finalize() {
25     releaseInstance();
26     --instanceCount;
27     if (instanceCount == 0) releaseClass();
28   }
29
30   private native void setupInstance(int width, int height);
31
32   private native void releaseInstance();
33
34   private static native void setupClass(Float3 GS_WEIGHTS, Float2Array INTENSITIES,
35                                         Byte4Array COLORS);
36
37   private static native void releaseClass();
38 }
```

B.4.2. Código C/C++

.h

```
#ifndef _POSTERIZEFN_H_
#define _POSTERIZEFN_H_
#define LOG_TAG "PosterizeFN"
#include <jni.h>
#include <stdio.h>
#include <fancier.h>
#include <fancier/internal/snippets.inc>
typedef struct PosterizeFN {
    jint width;
    jint height;
} PosterizeFN;

typedef struct PosterizeStaticFN {
    fcFloat3 GS_WEIGHTS;
    fcFloat2Array* INTENSITIES;
    fcByte4Array* COLORS;
} PosterizeStaticFN;

extern jclass PosterizeFN_class;
extern PosterizeStaticFN* selfStatic;

fcError PosterizeFN_run (PosterizeFN* self, fcRGBAImage* input,
                         fcRGBAImage* output);

#endif
```

.c

```
#include "../include/PosterizeFN.h"
#include <jni.h>
#include <stdio.h>
#include <fancier.h>
#include <fancier/internal/snippets.inc>
jclass PosterizeFN_class = NULL;
PosterizeStaticFN* selfStatic = NULL;
FC_JAVA_INSTANCE_HANDLERS(PosterizeFN);

fcError PosterizeFN_run(PosterizeFN* self, fcRGBAImage* input, fcRGBAImage* output){
  int __paralldroid_dims = 2;
  int __paralldroid_index[__paralldroid_dims];
  int __paralldroid_sz[] = { input->dims.x, input->dims.y} ;
  fcError err;
  for (__paralldroid_index[0] = 0;
       __paralldroid_index[0] < __paralldroid_sz[0]; ++__paralldroid_index[0]) {
    for (__paralldroid_index[1] = 0;
         __paralldroid_index[1] < __paralldroid_sz[1]; ++__paralldroid_index[1]) {
      jint x = (jint) __paralldroid_index[0];
      jint y = (jint) __paralldroid_index[1];
      {
        jint stage;
        for (stage = 0; stage < selfStatic->INTENSITIES->len; ++stage) {
          fcFloat2 intensity =
                fcFloat2Array_get(selfStatic->INTENSITIES, stage, (&err));
          fcByte4 color = fcByte4Array_get(selfStatic->COLORS, stage, (&err));
          fcByte4 pixel = fcRGBAImage_get(input, x, y, (&err));
          jfloat pixel_intensity = fcFloat3_dot(fcFloat3_divkf(
                fcByte3_convertFloat3(fcByte4_asByte3(pixel)), 255.0f),
```

```
                selfStatic->GS_WEIGHTS);
          if ((pixel_intensity <= intensity.y) && (pixel_intensity >= intensity.x))
            err = fcRGBAImage_set(output, x, y, color);
          }
        }
      }
  }

  return err;
}

JNIEXPORT void JNICALL
Java_ull_..._PosterizeFN_releaseClass (JNIEnv* env, jclass cls) {
  if (PosterizeFN_class)
    FC_FREE_CLASS_REF(env, PosterizeFN_class);

  free(selfStatic);
  selfStatic = NULL;
}

JNIEXPORT void JNICALL
Java_ull_..._PosterizeFN_releaseInstance (JNIEnv* env, jobject obj) {
  PosterizeFN* self = PosterizeFN_getJava(env, obj);
  if (self)
    PosterizeFN_freeJava(env, obj);

}

JNIEXPORT void JNICALL
Java_ull_..._PosterizeFN_setupClass (JNIEnv* env, jclass cls,
        jobject GS_WEIGHTSParam, jobject INTENSITIESParam, jobject COLORSParam) {
  if (!PosterizeFN_class)
    FC_INIT_CLASS_REF(env,
         "ull/hpc/pcg/paralldroid/linux/littletests/generated/PosterizeFN",
         PosterizeFN_class, "PosterizeFN.setupClass", FC_VOID_EXPR);

  selfStatic = calloc(1, sizeof(PosterizeStaticFN));
  fcError err;
  selfStatic->GS_WEIGHTS = fcFloat3_unwrap(env, GS_WEIGHTSParam, (&err));
  if (err != FC_EXCEPTION_SUCCESS)
    FC_EXCEPTION_HANDLE_ERROR(env, err, "fcFloat3_unwrap:selfStatic->GS_WEIGHTS",
                              FC_VOID_EXPR);

  selfStatic->INTENSITIES = fcFloat2Array_getJava(env, INTENSITIESParam);
  err = fcFloat2Array_syncToHost(selfStatic->INTENSITIES);
  FC_EXCEPTION_HANDLE_ERROR(env, err,
               "fcFloat2Array_syncToHost:selfStatic->INTENSITIES", FC_VOID_EXPR);
  selfStatic->COLORS = fcByte4Array_getJava(env, COLORSParam);
  err = fcByte4Array_syncToHost(selfStatic->COLORS);
  FC_EXCEPTION_HANDLE_ERROR(env, err, "fcByte4Array_syncToHost:selfStatic->COLORS",
                            FC_VOID_EXPR);
}

JNIEXPORT void JNICALL
Java_ull_..._PosterizeFN_setupInstance__II (JNIEnv* env, jobject obj,
                                            jint widthParam, jint heightParam) {
  PosterizeFN* self = PosterizeFN_getJava(env, obj);
  if (!self)
    self = PosterizeFN_allocJava(env, obj);

  FC_EXCEPTION_HANDLE_NULL(env, self, FC_EXCEPTION_INVALID_THIS,
                           "PosterizeFN.setupInstance", FC_VOID_EXPR);
  fcError err;
```

```
  self->width = widthParam;
  self->height = heightParam;
}

JNIEXPORT void JNICALL
Java_ull_..._PosterizeFN_run__... (JNIEnv* env, jobject obj, jobject inputParam,
                                   jobject outputParam) {
  PosterizeFN* self = PosterizeFN_getJava(env, obj);
  FC_EXCEPTION_HANDLE_NULL(env, self, FC_EXCEPTION_INVALID_THIS, "PosterizeFN.run",
                           FC_VOID_EXPR);
  fcError err;
  fcRGBAImage* __paralldroid_input = fcRGBAImage_getJava(env, inputParam);
  if (!fcRGBAImage_valid(__paralldroid_input)) {
    fcException_throwNative(env, __FILE__, __LINE__, "fcRGBAImage_valid",
                            FC_EXCEPTION_BAD_PARAMETER);
    return FC_VOID_EXPR;
  }

  err = fcRGBAImage_syncToHost(__paralldroid_input);
  FC_EXCEPTION_HANDLE_ERROR(env, err, "fcRGBAImage_syncToHost:__paralldroid_input",
                            FC_VOID_EXPR);
  fcRGBAImage* __paralldroid_output = fcRGBAImage_getJava(env, outputParam);
  if (!fcRGBAImage_valid(__paralldroid_output)) {
    fcException_throwNative(env, __FILE__, __LINE__, "fcRGBAImage_valid",
                            FC_EXCEPTION_BAD_PARAMETER);
    return FC_VOID_EXPR;
  }

  err = fcRGBAImage_syncToHost(__paralldroid_output);
  FC_EXCEPTION_HANDLE_ERROR(env, err, "fcRGBAImage_syncToHost:__paralldroid_output",
                            FC_VOID_EXPR);
  err = PosterizeFN_run(self, __paralldroid_input, __paralldroid_output);
  FC_EXCEPTION_HANDLE_ERROR(env, err, "PosterizeFN.run:PosterizeFN_run",
                            FC_VOID_EXPR);
}
```

B.5. Código Fancier OpenCL generado

B.5.1. Código Java

```
1  public class PosterizeFOCL extends Posterize {
2    static {
3      System.loadLibrary("PosterizeFOCL-lib");
4    }
5    private long nativeInstancePtr = 0L;
6    private static int instanceCount = 0;
7
8    public PosterizeFOCL(String fancierInitPath, AssetManager assets, int width,
9                         int height) {
10     super(fancierInitPath, assets);
11     this.width = width;
12     this.height = height;
13     if (instanceCount <= 0) {
14       instanceCount = 0;
15       setupClass(GS_WEIGHTS, INTENSITIES, COLORS);
16     }
17     ++instanceCount;
18     setupInstance(width, height);
19   }
20
21   @Kernel()
22   public native void run(@Range() RGBAImage input, RGBAImage output);
23
24   protected void finalize() {
25     releaseInstance();
26     --instanceCount;
27     if (instanceCount == 0) releaseClass();
28   }
29
30   private native void setupInstance(int width, int height);
31
32   private native void releaseInstance();
33
34   private static native void setupClass(Float3 GS_WEIGHTS, Float2Array INTENSITIES,
35                                         Byte4Array COLORS);
36
37   private static native void releaseClass();
38 }
```

B.5.2. Código C/C++

.h

```
ifndef _POSTERIZEFOCL_H_
#define _POSTERIZEFOCL_H_
#define NUM_KERNELS 1
#define LOG_TAG "PosterizeFOCL"
#include <jni.h>
#include <stdio.h>
#include <fancier.h>
#include <fancier/internal/snippets.inc>
typedef struct PosterizeFOCL {
    jint width;
    jint height;
} PosterizeFOCL;

typedef struct PosterizeStaticFOCL {
    fcFloat3 GS_WEIGHTS;
    fcFloat2Array* INTENSITIES;
    fcByte4Array* COLORS;
} PosterizeStaticFOCL;

extern jclass PosterizeFOCL_class;
extern PosterizeStaticFOCL* selfStatic;
static void compileKernels (JNIEnv* env, char* kernelDir);

static void releaseKernels ();

fcError PosterizeFOCL_run (PosterizeFOCL* self, fcRGBAImage* input,
                           fcRGBAImage* output);

#endif
```

.c

```
#include "../include/PosterizeFOCL.h"
#include <jni.h>
#include <stdio.h>
#include <fancier.h>
#include <fancier/internal/snippets.inc>
static int s_init = 0;
static cl_program s_program;
static cl_kernel s_kernels[NUM_KERNELS];
jclass PosterizeFOCL_class = NULL;
PosterizeStaticFOCL* selfStatic = NULL;
FC_JAVA_INSTANCE_HANDLERS(PosterizeFOCL);
static void compileKernels (JNIEnv* env, jobject asset_manager) {
  if (++s_init > 1)
    return;

  cl_int err;
  s_program = fcOpenCL_compileKernelAsset(env, asset_manager, "",
                                          "PosterizeFOCL.cl", (&err));
  FC_EXCEPTION_HANDLE_BUILD(env, err, "fcOpenCL_compileKernelFile", s_program,
                            FC_VOID_EXPR);
  s_kernels[0] = clCreateKernel(s_program, "run", (&err));
  FC_EXCEPTION_HANDLE_ERROR(env, err, "clCreateKernel:run", FC_VOID_EXPR);
}

static void releaseKernels () {
```

```
  if (--s_init > 0)
    return;

  if (s_kernels[0])
    clReleaseKernel(s_kernels[0]);

  if (s_program)
    clReleaseProgram(s_program);

}

fcError PosterizeFOCL_run (PosterizeFOCL* self, fcRGBAImage* input,
                           fcRGBAImage* output) {
  size_t __paralldroid_sz[] = { input->dims.x, input->dims.y} ;
  fcError err;
  err = clSetKernelArg(s_kernels[0], 0, sizeof(cl_mem), (&input->pixels->ocl));
  if (err)
    return err;

  err = clSetKernelArg(s_kernels[0], 1, sizeof(fcInt2), (&input->dims));
  if (err)
    return err;

  err = clSetKernelArg(s_kernels[0], 2, sizeof(cl_mem), (&output->pixels->ocl));
  if (err)
    return err;

  err = clSetKernelArg(s_kernels[0], 3, sizeof(fcInt2), (&output->dims));
   if (err)
    return err;

  err = clSetKernelArg(s_kernels[0], 4, sizeof(jint), (&self->width));
  if (err)
    return err;

  err = clSetKernelArg(s_kernels[0], 5, sizeof(jint), (&self->height));
  if (err)
    return err;

  err = clSetKernelArg(s_kernels[0], 6, sizeof(fcFloat3),(&selfStatic->GS_WEIGHTS));
  if (err)
    return err;

  err = fcFloat2Array_syncToDevice(selfStatic->INTENSITIES);
  if (err)
    return err;

  err = clSetKernelArg(s_kernels[0], 7, sizeof(cl_mem),
                       (&selfStatic->INTENSITIES->ocl));
  if (err)
    return err;

  err = clSetKernelArg(s_kernels[0], 8,sizeof(int),(&selfStatic->INTENSITIES->len));
  if (err)
    return err;

  err = fcByte4Array_syncToDevice(selfStatic->COLORS);
  if (err)
    return err;

  err = clSetKernelArg(s_kernels[0], 9, sizeof(cl_mem), (&selfStatic->COLORS->ocl));
  if (err)
    return err;
```

```
89
90    err = clSetKernelArg(s_kernels[0], 10, sizeof(int), (&selfStatic->COLORS->len));
91    if (err)
92      return err;
93
94    err = clEnqueueNDRangeKernel(fcOpenCL_rt.queue, s_kernels[0], 2, NULL,
95                                 __paralldroid_sz, NULL, 0, NULL, NULL);
96    if (err)
97      return err;
98
99    return FC_EXCEPTION_SUCCESS;
100 }
101
102 JNIEXPORT void JNICALL
103 Java_ull_..._PosterizeFOCL_releaseClass (JNIEnv* env, jclass cls) {
104   if (PosterizeFOCL_class)
105     FC_FREE_CLASS_REF(env, PosterizeFOCL_class);
106
107   releaseKernels();
108   free(selfStatic);
109   selfStatic = NULL;
110 }
111
112 JNIEXPORT void JNICALL
113 Java_ull_..._PosterizeFOCL_releaseInstance (JNIEnv* env, jobject obj) {
114   PosterizeFOCL* self = PosterizeFOCL_getJava(env, obj);
115   if (self)
116     PosterizeFOCL_freeJava(env, obj);
117
118 }
119
120 JNIEXPORT void JNICALL
121 Java_ull_..._PosterizeFOCL_setupClass (JNIEnv* env, jclass cls, jobject assetsParam,
122           jobject GS_WEIGHTSParam, jobject INTENSITIESParam, jobject COLORSParam) {
123   if (!PosterizeFOCL_class)
124     FC_INIT_CLASS_REF(env, "ull/.../test/PosterizeFOCL", PosterizeFOCL_class,
125                       "PosterizeFOCL.setupClass", FC_VOID_EXPR);
126
127   compileKernels(env, assetsParam);
128   selfStatic = calloc(1, sizeof(PosterizeStaticFOCL));
129   fcError err;
130   selfStatic->GS_WEIGHTS = fcFloat3_unwrap(env, GS_WEIGHTSParam, (&err));
131   if (err != FC_EXCEPTION_SUCCESS)
132     FC_EXCEPTION_HANDLE_ERROR(env, err, "fcFloat3_unwrap:selfStatic->GS_WEIGHTS",
133                               FC_VOID_EXPR);
134
135   selfStatic->INTENSITIES = fcFloat2Array_getJava(env, INTENSITIESParam);
136   err = fcFloat2Array_syncToHost(selfStatic->INTENSITIES);
137   FC_EXCEPTION_HANDLE_ERROR(env, err,
138                 "fcFloat2Array_syncToHost:selfStatic->INTENSITIES", FC_VOID_EXPR);
139   selfStatic->COLORS = fcByte4Array_getJava(env, COLORSParam);
140   err = fcByte4Array_syncToHost(selfStatic->COLORS);
141   FC_EXCEPTION_HANDLE_ERROR(env, err, "fcByte4Array_syncToHost:selfStatic->COLORS",
142                             FC_VOID_EXPR);
143 }
144
145 JNIEXPORT void JNICALL
146 Java_ull_..._PosterizeFOCL_... (JNIEnv* env, jobject obj, jint widthParam,
147                                 jint heightParam) {
148   PosterizeFOCL* self = PosterizeFOCL_getJava(env, obj);
149   if (!self)
150     self = PosterizeFOCL_allocJava(env, obj);
151
```

```
152   FC_EXCEPTION_HANDLE_NULL(env, self, FC_EXCEPTION_INVALID_THIS,
153                            "PosterizeFOCL.setupInstance", FC_VOID_EXPR);
154   fcError err;
155   self->width = widthParam;
156   self->height = heightParam;
157 }
158
159 JNIEXPORT void JNICALL
160 Java_ull_..._PosterizeFOCL_run__... (JNIEnv* env, jobject obj, jobject inputParam,
161                                      jobject outputParam) {
162   PosterizeFOCL* self = PosterizeFOCL_getJava(env, obj);
163   FC_EXCEPTION_HANDLE_NULL(env, self, FC_EXCEPTION_INVALID_THIS,
164                            "PosterizeFOCL.run", FC_VOID_EXPR);
165   fcError err;
166   fcRGBAImage* __paralldroid_input = fcRGBAImage_getJava(env, inputParam);
167   if (!fcRGBAImage_valid(__paralldroid_input)) {
168     fcException_throwNative(env, __FILE__, __LINE__,
169                             "fcRGBAImage_valid", FC_EXCEPTION_BAD_PARAMETER);
170     return FC_VOID_EXPR;
171   }
172
173   err = fcRGBAImage_syncToDevice(__paralldroid_input);
174   FC_EXCEPTION_HANDLE_ERROR(env, err,"fcRGBAImage_syncToDevice:__paralldroid_input",
175                             FC_VOID_EXPR);
176   fcRGBAImage* __paralldroid_output = fcRGBAImage_getJava(env, outputParam);
177   if (!fcRGBAImage_valid(__paralldroid_output)) {
178     fcException_throwNative(env, __FILE__, __LINE__, "fcRGBAImage_valid",
179                             FC_EXCEPTION_BAD_PARAMETER);
180     return FC_VOID_EXPR;
181   }
182
183   err = fcRGBAImage_syncToDevice(__paralldroid_output);
184   FC_EXCEPTION_HANDLE_ERROR(env,err,"fcRGBAImage_syncToDevice:__paralldroid_output",
185                             FC_VOID_EXPR);
186   err = PosterizeFOCL_run(self, __paralldroid_input, __paralldroid_output);
187   FC_EXCEPTION_HANDLE_ERROR(env, err, "PosterizeFOCL.run:PosterizeFOCL_run",
188                             FC_VOID_EXPR);
189 }
```

B.5.3. Código OpenCL

```
__kernel void run (__global uchar4* input,
                   uint2 __paralldroidParam_input_dims,
                   __global uchar4* output,
                   uint2 __paralldroidParam_output_dims,
                   float3 __paralldroidField_GS_WEIGHTS,
                   int __paralldroidField_BLUR_RADIUS,
                   __global float* __paralldroidField_gauss_kernel,
                   int __paralldroidField_gauss_kernel_len,
                   int __paralldroidField_MEDIAN_RADIUS,
                   __global float2* __paralldroidField_INTENSITIES,
                   int __paralldroidField_INTENSITIES_len,
                   __global uchar4* __paralldroidField_COLORS,
                   int __paralldroidField_COLORS_len) {

  int x = (int) get_global_id(0);
  int y = (int) get_global_id(1);
  {
    int stage;
    for (stage = 0; stage < __paralldroidField_INTENSITIES_len; ++stage) {
      float2 intensity = __paralldroidField_INTENSITIES[stage];
      uchar4 color = __paralldroidField_COLORS[stage];
      uchar4 pixel = input[index_img(__paralldroidParam_input_dims, x, y)];
      float pixel_intensity = dot((convert_float3(as_uchar3(pixel)) / 255.0f),
                                  __paralldroidField_GS_WEIGHTS);

      if ((pixel_intensity <= intensity.y) && (pixel_intensity >= intensity.x))
        output[index_img(__paralldroidParam_output_dims, x, y)] = color;
    }
  }
}
```

Bibliografía

[1] Alejandro Acosta, Sergio Afonso, and Francisco Almeida. Extending Paralldroid for the automatic generation of OpenCL code. *Proceedings of the 4th International Workshop on OpenCL*, pages 1–3, 2016.

[2] Sergio Afonso and Francisco Almeida. Rancid: Reliable benchmarking on Android platforms. *IEEE Access*, 8:143342–143358, 2020.

[3] Sergio Afonso and Francisco Almeida. Fancier: A Unified Framework for Java, C, and OpenCL Integration. *IEEE Access*, 9:164570–164588, 2021.

[4] Sergio Afonso and Francisco Almeida. *Parallel Computing and Low-Power Architectures*. PhD thesis, Universidad de La Laguna, 2022.

[5] Alberto Cabrera, Vicente Blanco, and Francisco Almeida. *Energy Efficiency Analysis and Modelization in Heterogeneous and High Performance Systems*. PhD thesis, Universidad de La Laguna, 2022.

[6] Grupo de Computación de Altas Prestaciones (GCAP) de la Universidad de La Laguna (ULL). Fancier High-Performance API. `https://github.com/HPC-ULL/Fancier`.

[7] Grupo de Computación de Altas Prestaciones (GCAP) de la Universidad de La Laguna (ULL). Tradux. `https://github.com/HPC-ULL/tradux`, `https://github.com/HPC-ULL/paralldroid-refactor`.