UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA POLITECNICA SUPERIOR DE GANDIA

Grado en Ing. Sist. de Telecom., Sonido e Imagen

"Optimización de algoritmos de procesado de imagen en sistemas microprocesadores empleando el lenguaje de programación específico HALIDE"

TRABAJO FINAL DE GRADO

Autor:
VICENTE ORTIZ GONZÁLEZ

Tutor:
DR. ALFONO MARTÍNEZ GARCÍA

Tutor externo:
DR. FERNANDO DE LA TORRE
(CARNEGIE MELLON UNIVERSITY)

GANDIA, 2015

A Pepe Medina.

Resumen

Actualmente la optimización de algoritmos para el procesamiento de imágenes en sistemas microprocesadores requieren un alto coste de recursos humanos, siendo necesario profesionales altamente cualificados ya que es fundamental tener un buen conocimiento del funcionamiento de los sistemas microprocesadores así como de las funciones intrínsecas específicas de sus respectivas arquitecturas. Este elevado coste no puede ser asumido por pequeñas compañías o grupos de investigación con un número reducido de recursos humanos para ser competitiva en este sector. En este contexto surge el lenguaje de programación específico *HALIDE* desarrollado en el *Massachusetts Institute of Technology* (*MIT*) en 2012 para facilitar la codificación eficiente de algoritmos de procesado de imagen utilizando el lenguaje de programación C++. Actualmente permite compilaciones para arquitecturas *x86-64* con el uso de *SSE*, *ARMv7* con el uso de *NEON*, *CUDA* y *OpenCL*. Esto permitirá reducir el coste de recursos humanos necesarios para los desarrollos de dichos algoritmos.

Palabras clave: *visión artificial HALIDE C++ microprocesadores*

Abstract

Currently the optimization of algorithms for image processing in microprocessor systems require a high cost of human resources, highly qualified professionals still necessary because it is essential to have a good knowledge about microprocessor systems as well as the use of intrinsic function for each architecture. This high cost can not be assumed by small companies, or research groups with a small number of human resources, in order to be competitive in this sector. In this context were we find *HALIDE*, an specific programming language developed at the *Massachusetts Institute of Technology* (*MIT*) in 2012 to facilitate the efficient coding of image processing algorithms using the C ++ programming language. Currently *HALIDE* allows builds for *x86-64* architectures using the *SSE* extensions, *ARMv7* with *NEON*, *CUDA* and *OpenCL*. This language aims to reduce the cost of human resources required for the development of such algorithms.

Keywords: *computer vision HALIDE C++ microprocessor*

Índice de contenidos

Capítulo 1. Introducción al trabajo final de grado — *1*

1. Contexto — **1**
- 1.1 Lugar de desarrollo — 1
- 1.2 Programa PROMOE — 1
- 1.3 Human Sensing Laboratory — 2

2. Definición del problema a resolver — **2**
- 2.1 Punto de partida — 2
- 2.2 Contextos de aplicación — 3
- 2.3 Procesos de optimización — 3
- 2.4 Definición de la tecnología utilizada en el proyecto — 5
- 2.5 Planteamiento del problema — 6
- 2.6 Objetivos del trabajo final de grado — 6
- 2.7 Estructura de la memoria del trabajo final de grado — 7

Capítulo 2. Técnicas de eliminación de ruido basadas en métodos variacionales — *8*

1. Rudin, Osher y Fatemi (ROF) — **8**
- 1.1 Planteamiento Bayesiano — 8
  - 1.1.1 Máximo a posteriori — 9
  - 1.1.2 Adaptación a diferentes modelos de ruido — 10
- 1.2 Métodos resolutivos — 10

Capítulo 3. Sistemas microprocesadores — *11*

1. Aspectos técnicos a tener en cuenta en la programación — **11**
- 1.1 Paralelismo — 11
  - 1.1.1 A nivel de hilo — 11
  - 1.1.2 A nivel de datos — 11
- 1.2 Memoria volátil — 12
- 1.3 Instrucciones por segundo (IPS) — 12

2. Relación entre las prestaciones del sistema en el procesado de imágenes — **12**
- 2.1 Esquema de cálculo en la raíz: Sobrecarga de la memoria primaria — 12
- 2.2 Esquema de cálculo insertado: Sobrecarga del núcleo — 13
- 2.3 Esquema de cálculo paralelizado: Opción de compromiso — 14

Capítulo 4. Actuales herramientas de codificación — *15*

1. Librerías de visión artificial — **15**

2. Librerías de álgebra lineal — **16**
- 2.1 Evaluación de prestaciones — 17

3. Otras herramientas para una implementación optimizada — **17**

Capítulo 5. Lenguaje de programación específico HALIDE — *18*

1. Lenguaje embebido en C++ — **18**

2. Estructura básica del programa — **19**

3. Definición del algoritmo — **19**
- 3.1 Primitivas del lenguaje — 19

4. Definición del esquema — **20**
- 4.1 Esquemas de evaluación de una función — 20
  - 4.1.1 Orden por defecto — 21

4.1.2 Reordenación de variables 21
4.1.3 División de variables 22
4.1.4 Fusión de variables 22
4.1.5 Evaluación en mosaico 22
4.1.6 Evaluación en vectores 23
4.1.7 Desenrollar un bucle 24
4.1.8 Paralelizado 24
4.2 Esquema de cálculo multi-etapas 25
4.2.1 Orden por defecto 25
4.2.2 Evaluación en el inicio 26
4.2.3 Evaluación por filas 26
4.2.4 Evaluación por filas con búfer externo 27
4.2.5 Evaluación por columnas con buffer externo 28
4.2.6 Rendimiento de cada esquema 28

5. Modos de compilación 29
5.1.1 Compilación en tiempo de ejecución 29
5.1.2 Compilación estática 29

Capítulo 6. Sistemas microprocesadores embebidos basados en distribuciones Android/Linux 30

1. Introducción 30

2. Android y Linux 30

3. Modificaciones en el núcleo Vanilla 31
3.1 Términos de licencia 31

4. Entorno de usuario 32

5. Conclusiones relativas al proceso de optimización 32

Capítulo 7. Desarrollo del modelo de referencia del algoritmo 33

1. Descripción y simplificación del algoritmo 33

2. Simulación y resultados 33

Capítulo 8. Implementación del algoritmo mediante el uso de las técnicas habituales 35

1. Objetivo 35

2. Uso de librerías 36
2.1 Estructura de ficheros y proceso de compilación 36
2.2 Implementación con librería OpenCV 37
2.2.1 Codificación 37
2.2.2 Evaluación en GNU/Linux con arquitectura x86-64 37
2.2.3 Evaluación en Android/Linux con arquitectura ARMv7 38
2.2.4 Valoraciones 38
2.3 Implementación con librerías OpenCV y Eigen 39
2.3.1 Codificación 39
2.3.2 Evaluación en GNU/Linux con arquitectura x86-64 40
2.3.3 Evaluación en Android/Linux con arquitectura ARMv7 40
2.3.4 Valoraciones 40
2.4 Implementación con librerías OpenCV y Armadillo+OpenBLAS 41
2.4.1 Codificación 41
2.4.2 Evaluación en GNU/Linux con arquitectura x86-64 41
2.4.3 Evaluación en Android/Linux con arquitectura ARMv7 42
2.4.4 Valoraciones 42
2.5 Comparación de resultados 43

3. Uso de otros métodos habituales de optimización 43

Capítulo 9. Implementación del algoritmo utilizando el lenguaje HALIDE 45

1. Estructura del programa y modos de compilación ____ 45
2. Definición del algoritmo ____ 45
3. Definición del esquema ____ 46
3.1 Evaluación en línea ____ 46
3.2 Evaluación en la raíz ____ 47
3.3 Evaluación en mosaico con cambios en el orden de evaluación de las sub-funciones ____ 47
3.4 Evaluación por filas con cambios en el orden de evaluación de las sub-funciones ____ 48
4. Compilación cruzada ____ 49
5. Resultados ____ 49
5.1 Sistemas operativos GNU/Linux sobre arquitectura x86-64 ____ 50
5.2 Sistemas operativos Android/Linux sobre arquitectura ARMv7 ____ 50
6. Valoración de los diferentes esquemas ____ 51
6.1 Comportamiento de la localidad y la redundancia ____ 51
6.2 Comportamiento de la paralelización ____ 52
Capítulo 10. Conclusiones y líneas futuras ____ 53
1. Conclusiones ____ 53
2. Futuras líneas de investigación ____ 53
A. Referencias ____ 54
1. Referencias ____ 54
Anexo 1. Implementación en Matlab ____ 55
Anexo 2. Implementación con OpenCV ____ 56
Anexo 3. Implementación con Eigen ____ 57
Anexo 4. Implementación con Armadillo ____ 59
Anexo 5. Implementación con HALIDE ____ 61

Capítulo 1. Introducción al trabajo final de grado

En este capítulo se realiza la contextualización del trabajo a realizar en el proyecto final de grado, se muestran objetivos del mismo y la estructura de la memoria realizada.

1. Contexto

1.1 Lugar de desarrollo

El Trabajo Final de Grado que se presenta en esta memoria ha sido realizado por el alumno Vicente Ortiz González durante una estancia realizada desde Octubre de 2014 hasta la actualidad en el *Robotics Institute* (*RI*) de la universidad *Carnegie Mellon University* (*CMU*) situada en la ciudad de Pittsburgh (Pennsylvania, Estados Unidos de América).

La estancia a la que se hace referencia ha sido facilitada por la Oficina de Programas Internacionales de Intercambio (OPII) de la Universitat Politècnica de València (UPV) a través de la convocatoria de becas PROMOE para el curso 2014/2015

1.2 Programa PROMOE

El programa PROMOE se gestiona desde la Oficina de Programas Internacionales de Intercambio (OPII) de la UPV. Desde la propia página web de la entidad obtenemos la definición de este programa:

> << *Promoe es un programa propio de la UPV, es decir financiado íntegramente con fondos de la UPV, cuyo objetivo es establecer un programa de ayudas para el intercambio de estudiantes con universidades de EEUU, China, Canadá, América Latina, Corea, Australia o Japón entre otros con la que exista convenio de cooperación institucional e intercambio de estudiantes.*
>
> *Los estudiantes Promoe, podrán realizar parte de sus estudios en una Universidad de prestigio, durante un período de entre 4 y 10 meses, que les permitirá contactar con una cultura diferente, aprender o perfeccionar una lengua extranjera, experimentar diferentes metodologías de enseñanza, desarrollo personal y de formación, hacer amigos de diferentes países y culturas, a la vez de abrirles nuevas puertas dentro del mercado laboral. Todo ello hace que desde la UPV se anime a los estudiantes a completar su formación en cualquiera de las Universidades socias.* >>
>
> http://www.upv.es/entidades/OPII

Se trata de un programa que cuenta con acuerdos con diferentes universidades a nivel mundial para facilitar el intercambio de estudiantes con la UPV y que, además, cuenta con una bolsa económica financiada con fondos públicos de manera que, en la medida de lo posible, cualquier estudiante pueda realizar una estancia en una universidad extranjera independientemente de su situación económica.

Habiendo sido beneficiario de esta ayuda, y tras haber cursado unos estudios superiores en una institución pública, cabe dedicar estas líneas de gratitud al sistema público de educación que me ha permitido desarrollarme profesionalmente independientemente de la situación económica y social de mi familia. Con estas humildes líneas agradecer a todas las personas que luchan y contribuyen en la defensa de nuestro sistema público de educación.

1.3 HUMAN SENSING LABORATORY

El presente trabajo se ha realizado en el seno del *Human Sensing Laboratory* [3] liderado por el Dr. Fernando de la Torre. El objetivo de este departamento es modelar y entender el comportamiento humano desde información obtenida mediante diferentes tipos de sensado: visión, audio, parámetros biológicos, etc. El trabajo de este departamento se centra en aplicaciones en el campo de la salud, visión artificial, biometría e interfaces humano-máquina. La investigación de este laboratorio se financia a través de agencias federales (*NSF*, *NIMH*, *DARPA* y *ONR*), así como a través de la industria privada.

Dentro de esta área de trabajo el *Human Sensing Lab* cuenta con un gran prestigio, reconocido a nivel mundial, en el campo de la investigación en visión artificial. Es dentro de este campo el contexto en el que se desarrolla este trabajo.

2. DEFINICIÓN DEL PROBLEMA A RESOLVER

2.1 PUNTO DE PARTIDA

Como bien es sabido, la tarea de la investigación tiene como finalidad aportar conocimiento a la comunidad científica y, a partir de este conocimiento, generar nuevas herramientas que contribuyan a una mejora de la sociedad en su conjunto (bien a través del desarrollo de un nuevo producto, bien a través de la introducción de mejoras en procesos productivos, etc.). No obstante en muchas ocasiones el paso entre la publicación de un artículo científico y su puesta en práctica de una manera útil para la sociedad resulta bastante laborioso.

Dentro del área de la visión artificial nos encontraríamos un proceso que, en un punto de partida, consiste en crear un algoritmo matemático que resuelva el problema dado (como por ejemplo detectar una cara en un flujo de vídeo). Es probable que en esta etapa de desarrollo se estén utilizando herramientas de simulación matemática (por ejemplo *Matlab*). Estas herramientas de simulación nos permiten un rápido prototipado del sistema que estamos desarrollando, sin embargo no nos suelen permitir una conversión directa de la simulación realizada en una implementación funcional del sistema. Es decir, por normal general no disponemos de un botón en las aplicaciones de simulación que nos implementen el sistema diseñado en un microcontrolador, en una *FPGA*, en un sistema *software-radio*, en un ejecutable, etc. de una manera eficiente y optimizada a los requerimientos del contexto de aplicación del sistema. Generalmente se hace necesario contar con profesionales experimentados en la tecnología de implementación para que, tras meses de trabajo, consigan una aplicación práctica funcional del sistema ideado.

En nuestro caso vamos a focalizarnos en un proyecto tipo de visión artificial. La visión artificial se basa en el tratamiento de la información obtenida a través de un sensor que, de manera general, captará las reflexiones de ciertas longitudes de onda del espectro visible producidas sobre un área determinada del entorno. Al tratarse de un tipo sensado es muy probable que la tecnología forme parte de otros sistemas y que deba implementarse de manera embebida.

Por sistema embebido, o integrado, entendemos que se trata de un sistema desarrollado para cubrir expresamente las necesidades de la aplicación específica. Por el lado contrario encontraríamos un sistema de computación de propósito general en el que se dispone de una amplia gama de dispositivos integrados que nos permiten la ejecución de diversas aplicaciones diferentes (desde una aplicación de telemetría – *haciendo uso de una tarjeta de adquisición de datos y de la tarjeta de red* –, hasta una aplicación de computación distribuida – *utilizando de manera intensiva el procesador y transmitiendo los resultados a través de la tarjeta de red*).

También es frecuente que el sistema de visión artificial desarrollado no forme un sistema en sí mismo, sino que la tecnología desarrollada forme parte de otros sistemas que a su vez serán embebidos. Teniendo en cuenta todo lo anterior se obtiene la conclusión de que se hace

necesaria una implementación muy eficiente y bien optimizada que nos permita la ejecución de la aplicación en sistemas embebidos, bien de manera independiente o compartida, que en su mayoría ofrecerán unas prestaciones técnicas por debajo de lo que se desearía idealmente.

El desarrollo del sistema embebido podría realizarse mediante el uso de diferentes tecnologías, pero en nuestro caso vamos a focalizarnos en sistemas microprocesadores ya que esta es la línea de preferencia del laboratorio en el que me encuentro debido a que, según se afirma:

- Los sistemas microprocesador actuales ofrecen una *capacidad de computación* que resuelve las necesidades de los algoritmos de visión de una manera aceptable.
- El *tiempo de desarrollo* de una aplicación en un sistema microprocesador es mucho menor al tiempo de desarrollo necesario para la implementación en otras tecnologías como sistemas de lógica programable.

Llegados a este punto podemos resumir todo lo expuesto anteriormente y definir lo que sería el proceso de desarrollo de un proyecto tipo de visión artificial en las siguientes etapas:

1. **Planteamiento del problema**. Definición del problema que se pretende resolver.
2. **Investigación**. Estudio del estado del arte y conocimiento de la literatura disponible, así como de problemas similares en otras áreas del conocimiento. Definición de la resolución del problema.
3. **Simulación**. Puesta en práctica de la resolución del problema y validación de las conjeturas. Obtención del modelo de referencia.
4. **Implementación**. Desarrollo de la solución al problema validada previamente mediante el uso de una tecnología específica. Validación de la aplicación realizada mediante la comparación con el modelo de referencia.
5. **Optimización**. Adaptación de la implementación realizada a los requerimientos del contexto de aplicación y a las prestaciones de la tecnología disponible. Validación de la optimización realizada mediante la comparación con el modelo de referencia, comprobando que no se ha modificado el comportamiento esperado.
6. **Aplicación**. Se dispone de un sistema útil preparado para ser utilizado de manera eficiente.

2.2 CONTEXTOS DE APLICACIÓN

El contexto de aplicación nos definirá los requerimientos del sistema y condicionará las prestaciones ofrecidas por la tecnología utilizada. Estas circunstancias externas al propio algoritmo nos definirán la manera de optimizar la implementación. También puede suceder que incluso en un mismo proyecto, a medida que se avanza en el desarrollo del sistema, se sufran modificaciones en las características de la tecnología utilizada. Debido a factores de diversa índole, como el coste o la disponibilidad del producto, podemos encontrarnos con un sistema microprocesador inicial que cuente con escasos recursos de memoria volátil y alta capacidad de cómputo, pero al cabo de un tiempo sufrir modificaciones en el hardware y disponer en última instancia de un sistema microprocesador con gran capacidad y ancho de banda de memoria volátil pero con menos capacidad de cómputo.

2.3 PROCESOS DE OPTIMIZACIÓN

El proceso de optimización para algoritmos de complejidad elevada no resulta trivial, y a menudo se traduce en una nueva implementación del algoritmo. Como se describirá más adelante este proceso requiere de la utilización de librerías de terceros, como por ejemplo:

- ***OpenCV***. Es una librería que incorpora una gran cantidad de funciones utilizadas en visión artificial implementadas de manera optimizada.

- ***Eigen***. Es una librería en C++ para cálculo de álgebra lineal.
- ***Armadillo***. Otra librería en C++ para cálculo de algebra lineal.
- ***GNU Scientific Library* (*GSL*)**. Es una librería de cálculo numérico en C y C++.
- ***OpenBLAS***. Una implementación de *Basic Linear Algebra Subprograms* (*BLAS*), que es una especificación que define un conjunto de rutinas para el cálculo de operaciones de álgebra lineal.
- ***Automatically Tuned Linear Algebra Software* (*ATLAS*)**. Optimiza las operaciones definidas en las especificaciones *BLAS* de manera automática, como por ejemplo adecuando el tamaño de bloques al tamaño de la memoria caché del microprocesador.

En el caso de la utilización de librerías de terceros hay que valorar la disponibilidad de las librerías optimizadas para cada tipo de arquitectura de procesador, lo cual no siempre suele ocurrir.

Por otro lado se hace necesario un conocimiento de la arquitectura sobre la que se está trabajando así como otras especificaciones como la memoria volátil disponible y el ancho de banda para el acceso a ella. Estas especificaciones nos condicionarán la estructura del programa indicándonos si debemos hacer uso de buffers locales dentro de bucles, si es más óptimo computar muestras de manera redundante y evitar accesos a memoria, si debemos vectorizar el cálculo, si disponemos de varios núcleos para paralelizar y crear un pool de subprocesos, etc. Todo esto haciendo uso de funciones intrínsecas, las cuales son funciones propias del lenguaje de programación que interpreta el compilador y que parte de ellas se corresponden directamente con instrucciones *SIMD* según la arquitectura del microprocesador:

- Instrucciones *SSE* para arquitecturas *x86-64*
- Instrucciones *NEON* para arquitecturas *ARM*

Como ejemplo para ilustrar este arduo proceso citamos [7] al que hace referencia el equipo de trabajo del lenguaje de programación específico *HALIDE* (del cuál hablaremos más adelante). Para una implementación de un filtro laplaciano tenemos:

- Implementación de referencia en C++: 300 líneas de código.
- Implementación realizada por el personal de la compañía Adobe:
  - 3 meses de trabajo.
  - 1500 líneas de código.
  - 10 veces más rápido que la implementación de referencia.
  - Uso de técnicas como:
    - Paralelización.
    - Vectorización.
    - Organización adecuada del cómputo de los datos.
    - Etc.

Esto nos demuestra, mediante un ejemplo de un caso real de la industria, que el proceso de optimización para unas especificaciones de un sistema microprocesador determinado es un proceso muy costoso en términos de horas-persona. Este coste puede darse la situación de que pueda ser asumido debido a una elevada facturación y una disponibilidad de plantilla, pero puede darse la situación (como en pequeños grupos de investigación y pequeñas y medianas empresas) que el hecho de dedicar el trabajo de un miembro del equipo durante 3 meses a optimizar una implementación sea inasumible. Además se hace necesario de personal altamente cualificado que tenga conocimiento de las técnicas de programación, del uso de funciones intrínsecas, de la existencia de librerías de terceros, etc.

Otro aspecto es que previamente a la codificación hay que diseñar la estructura del programa y sobre este diseño se codificará. Pueden darse casos de elegir cierta estructura de programa y que, una vez programado el algoritmo, se demuestre que quizás esa estructura no era la más adecuada, por lo que se debería de invertir tiempo nuevamente en la confección de un nuevo diseño y la codificación de este. Por lo que otro factor limitante, y que condicionará el

resultado del proceso de optimización, es la elección de la estructura de programa realizada en el punto de partida.

Finalmente cabe reseñar la existencia de otro factor fundamental, que escapa al proceso propio de la codificación pero que resulta necesario en el proceso de optimización, y es el conocimiento en profundidad de las herramientas matemáticas utilizadas en la definición del algoritmo. Cabe siempre preguntarse, como se verá más adelante, si la definición del algoritmo que se desea implementar es la más adecuada o, si por el contrario, encontramos otras rutinas que nos permitan llegar al mismo resultado explotando al máximo las características técnicas del sistema microprocesador objetivo mediante el uso de las herramientas matemáticas.

2.4 DEFINICIÓN DE LA TECNOLOGÍA UTILIZADA EN EL PROYECTO

Se plantea la siguiente arquitectura sobre la que se va a trabajar en el presente proyecto y que representa un proyecto tipo de visión artificial:

- **Sensor de imagen.** Generalmente un sensor *CMOS*. En nuestro caso de prototipado se utilizará una cámara *USB* conectada al sistema microprocesador, o bien, ya disponible de manera embebida en el sistema microprocesador (teléfonos inteligentes). Si bien para las pruebas de rendimiento no se utilizará ya que se trabajará sobre imágenes en fichero.
- **Sistema microprocesador.** Se utilizarán tres tipos de sistema microprocesador diferentes que difieren en prestaciones y arquitectura, lo cual nos ilustrará la importancia de la etapa de optimización:
    - Sistemas microprocesadores embebidos:
        - Placa de desarrollo *NVIDIA Jetson TK1*. [12]
            - Procesador Tegra K1 (versión de 32 bits) de 2 núcleos a 2’3MHz, memoria caché L1/L2 de tamaño 32KB/2MB.
            - Memoria volátil primaria con capacidad de 2GB.
        - Teléfono inteligente *Samsung Galaxy S4*.
            - Procesador de 2 núcleos a 1’3MHz, memoria caché L1/L2 de tamaño 16KB/512KB por núcleo
            - Memoria volátil primaria con capacidad de 1GB.
        - Teléfono inteligente *Samung Galaxy SII.*
            - Procesador de 4 núcleos a 1’9MHz, memoria caché L1/L2 de tamaño 16KB/2MB por núcleo
            - Memoria volátil primaria con capacidad de 2GB.
    - Sistemas microprocesadores de propósito general:
        - Ordenador portátil *ASUS* A55V con procesador *Intel Core* i7 – 3610QM.
            - Procesador de 8 núcleos a 2’3MHz, memoria caché L1/L2/L3 de tamaño 32KB/256KB/6MB.
            - Memoria volátil primaria con capacidad de 16GB.
- **Representación de los datos.** Tratamiento de la información obtenida. En nuestro caso se utilizará un monitor conectado a una salida de video para mostrar el efecto del algoritmo implementado sobre la imagen (o en teléfonos inteligente mediante la visualización de la imagen en la pantalla incorporada).

Por lo que respecta a la parte de software se utilizará el lenguaje de programación C++ que se compilará en:

- Máquina con sistema operativo *GNU/Linux*, distribución *Ubuntu* 14.04.
- Compilador *GNU Compiler Collection* (*GCC*), versión 4.7

Para el caso de sistemas embebidos se utiliza en todos ellos sistemas operativos basados en máquinas virtuales de *Java* (distribuciones de Android/Linux) se utilizara también el lenguaje C++ para el desarrollo de los algoritmos haciendo uso de librerías compartidas. Estas librerías

podrán ser utilizadas a través de la infraestructura digital *Java Native Interface* (*JNI*) que nos permitirá el uso del código optimizado.

2.5 PLANTEAMIENTO DEL PROBLEMA

Llegados a este punto tenemos conocimiento de cómo es el trabajo que se realiza para el desarrollo práctico de un algoritmo de visión artificial sobre sistemas microprocesadores, es decir, el trabajo que conduce a la obtención de un producto semi-desarrollado útil para la industria. Con el término semi-desarrollado nos referimos al hecho de que se dispone de un sistema funcional y eficiente utilizando las tecnologías *de facto* de la industria. Esto ayuda enormemente a la colocación de la tecnología desarrollada ya que facilita la labor de integración en otros sistemas y reduce en gran medida la inversión en desarrollo que debería emplear la empresa receptora de la tecnología.

Por lo que se ha visto en el apartado dedicado a los procesos de optimización nos hacemos una idea de los costes que implican adaptar nuestra implementación a los diferentes tipos de arquitecturas y sistemas microprocesadores que existen en el mercado. En este contexto es en el que nace el lenguaje de programación específico *HALIDE*. [8]

HALIDE es un nuevo lenguaje de programación diseñado para facilitar la implementación de algoritmos de visión artificial de una manera eficiente. Es un lenguaje de programación específico embebido en C++ y actualmente cubre una amplia gama de arquitecturas como x86-64, ARMv7 y CUDA, entre otras.

El equipo de desarrollo de este lenguaje se encuentra en el Instituto Tecnológico de Massachussetts (*MIT*) y cuenta con profesionales altamente cualificados en el campo de la computación y la visión artificial. Según los creadores, este lenguaje de programación específico permite la implementación eficiente de algoritmos de visión en un tiempo realmente reducido y con unos resultados espectaculares. Se ofrecen varias publicaciones y se exponen casos particulares que demuestran las afirmaciones con resultados. Exponen, entre otros, el caso anteriormente comentado sobre el filtro laplaciano y se obtienen con *HALIDE* los siguientes resultados:

- Implementación utilizando *HALIDE*:
    - 1 día de trabajo (frente a los 3 meses empleados en Adobe).
    - 20 veces más rápido que la implementación de referencia.
    - 2 veces más rápido que la implementación de Adobe.

A la vista de los prometedores resultados que ofrece el nuevo lenguaje específico de programación *HALIDE* se plantea al estudiante la resolución del siguiente problema que es el objetivo principal del Trabajo Final de Grado objeto de la presente memoria:

¿Es HALIDE una tecnología útil para que sea adoptada en el laboratorio?

Para ello se solicita al estudiante que implemente un sencillo algoritmo de visión artificial utilizando la nueva tecnología para diferentes arquitecturas y sistemas microprocesadores. Este algoritmo será un filtro para la eliminación de ruido basado en el concepto de variación total, el cual suele ser bastante costoso en términos computacionales.

2.6 OBJETIVOS DEL TRABAJO FINAL DE GRADO

El objetivo principal de este trabajo final de grado consiste en la evaluación del lenguaje de programación específico HALIDE mediante la implementación de un algoritmo de visión artificial en diversos sistemas microprocesadores con diferentes características técnicas.

Para ello este trabajo final de grado se descompone en los siguientes objetivos específicos:

- Conocer las técnicas de eliminación de ruido en imágenes basadas en métodos variacionales.
- Estudio del algoritmo de eliminación de ruido presentado en la publicación [15].

- Conocer las herramientas disponibles en la actualidad para la implementación de algoritmos de visión artificial en leguaje C++ de una manera optimizada.
- Estudiar el lenguaje de programación HALIDE.
- Conocer la relación entre los esquemas de procesado de imágenes y las prestaciones de los sistemas microprocesadores.
- Conocer las particularidades de los sistemas microprocesadores embebidos con sistema operativo Android/Linux.
- Implementar el algoritmo utilizando las técnicas habituales de programación y comparar el proceso con la implementación de HALIDE.

2.7 ESTRUCTURA DE LA MEMORIA DEL TRABAJO FINAL DE GRADO

Los objetivos de la memoria del trabajo final de grado señalados en el punto anterior se desarrollaran en la memoria siguiendo la estructura que se muestra a continuación.

- **Capítulo 1. Introducción al trabajo final de grado**.
  En este capítulo se realiza la contextualización del trabajo a realizar en el proyecto final de grado, se muestran objetivos del mismo y la estructura de la memoria realizada.
- **Capítulo 2. Técnicas de eliminación de ruido basadas en métodos variacionales**.
  Se introduce en este capítulo un algoritmo de eliminación de ruido basado presentado por Leonid I. Rudin, Stanley Osher y Emad Fatemi en la publicación “Nonlinear total variation based noise removal algorithms” en 1992.
- **Capítulo 3. Sistemas microprocesadores.**
  En este capítulo se realizará un breve análisis de las características técnicas de los sistemas microprocesadores más importantes a tener en cuenta en la implementación optimizada de algoritmos de visión artificial.
- **Capítulo 4. Actuales herramientas de codificación.**
  En este capítulo se analizan las diferentes herramientas y técnicas de programación disponibles en la actualidad para la implementación de algoritmos de procesado de imagen en lenguaje C++.
- **Capítulo 5. Lenguaje específico HALIDE.**
  En este capítulo se presenta este lenguaje de programación HALIDE explicando de manera resumida su estructura interna así como sus primitivas, esquemas de optimización, modos de compilación, etc.
- **Capítulo 6. Sistemas microprocesadores embebidos basados en distribuciones Android/Linux.**
  En este capítulo se explica la arquitectura de una distribución Android/Linux desde un particular punto de vista que nos ayudará al desarrollo de código nativo en estas plataformas.
- **Capítulo 7. Desarrollo del modelo de referencia del algoritmo.**
  En este capítulo se realizará, utilizando Matlab, la simulación de un algoritmo de eliminación de ruido basado en el concepto de variación total.
- **Capítulo 8. Implementación del algoritmo mediante el uso de las técnicas habituales.**
  En este capítulo se implementará un algoritmo de eliminación de ruido basado en el concepto de variación total utilizando las técnicas de programación habituales.
- **Capítulo 9. Implementación del algoritmo utilizando el lenguaje HALIDE.**
  En este capítulo se implementará un algoritmo de eliminación de ruido basado en el concepto de variación total utilizando el lenguaje de programación HALIDE.
- **Capítulo 10. Conclusiones y líneas futuras.**
  En este capítulo se comparan las diferentes implementaciones realizadas obteniendo las conclusiones acerca del uso del nuevo lenguaje de programación HALIDE. Se trazan las futuras líneas de investigación que sería conveniente seguir para llegar a conclusiones más sólidas acerca del beneficio de utilizar HALIDE como lenguaje de preferencia.

Capítulo 2. Técnicas de eliminación de ruido basadas en métodos variacionales

Se introduce en este capítulo un algoritmo de eliminación de ruido basado presentado por Leonid I. Rudin, Stanley Osher y Emad Fatemi en la publicación "Nonlinear total variation based noise removal algorithms" en 1992. Se realizará un enfoque haciendo uso de la estadística bayesiana para facilitar la comprensión conceptual del funcionamiento del algoritmo.

1. RUDIN, OSHER Y FATEMI (ROF)

Cuando en la literatura se hace referencia al método *ROF* para la eliminación de ruido se está hablando del método original planteado por Leonid I. Rudin, Stanley Osher y Emad Fatemi en la publicación realizada en 1992 [15] sobre algoritmos no-lineales para la eliminación de ruido basados en la variación total.

Una técnica básica para la eliminación de ruido, como bien conocemos los ingenieros de telecomunicaciones, es la técnica de promediado. Está no es que sea una buena técnica para la eliminación de ruido en señales constantes, sino que es la mejor como es demostrable. Sin embargo esto no sucede para las señales que sufren variaciones en el tiempo, es decir, que conllevan componentes frecuenciales en el intervalo observado. Desde el punto de vista de una señal binaria *NRZ* unipolar observaríamos un ensanchamiento de los pulsos tras haberla sometido a un promediado.

En el caso de imágenes nos va a suceder el mismo efecto pero con un impacto mucho mayor. Esto es debido a que la información más importante que contiene una imagen se encuentra en los contornos de los objetos.

En este sentido se presenta en la publicación mencionada un método para la eliminación de ruido de manera no-invasiva y manteniendo los contornos en la imagen.

1.1 PLANTEAMIENTO BAYESIANO

El fundamento del método presentado debería explicarse mediante el análisis funcional. Bien es cierto también que se le puede dar un enfoque Bayesiano el cual, desde mi punto de vista, resulta más *visual* para el lector no especializado.

Vamos a considerar una función de una variable $r(x)$, que representará la señal observada. Podemos asumir que esta función es igual a la señal original más el ruido aditivo:

$$r(x) = u(x) + \eta(x)$$

Donde $u(x)$ representa la señal original y $\eta(x)$ vamos a asumir en un primer momento que representa un ruido blanco, aditivo y Gaussiano (*AWGN*) con una varianza σ^2.

El ruido considerado presenta una distribución Gaussiana. Una variable aleatoria X con distribución Gaussiana presenta una función de densidad de probabilidad de la forma:

$$f_X(x) = \frac{1}{\sigma_X\sqrt{2\pi}} e^{\left[-\frac{(x-m_X)^2}{2\sigma_X^2}\right]}$$

Donde m_X representa la media de los valores de X y σ^2 representa la varianza.

Por lo que cada valor disponible de $r(x)$ tendrá una probabilidad de representar el valor original $u(x)$:

$$p\big(r(x)\big|u(x)\big) = \frac{1}{\sigma\sqrt{2\pi}} e^{\left[-\frac{\left(r(x)-m_{r(x)}\right)^2}{2\sigma^2}\right]} = \frac{1}{\sigma\sqrt{2\pi}} e^{\left[-\frac{\left(r(x)-u(x)\right)^2}{2\sigma^2}\right]}$$

O lo que es lo mismo para el conjunto de valores:

$$p(r|u) = \frac{1}{\sigma\sqrt{2\pi}} e^{\left[-\frac{\int_\Omega \left(r(x)-u(x)\right)^2 dx}{2\sigma^2}\right]}$$

Es decir, $p(r|u)$ representa la similitud entre el conjunto observado $r(x)$ y el conjunto original $u(x)$.

1.1.1 Máximo a posteriori

El objetivo es encontrar el conjunto de valores de $u(x)$ más probables, esto es que maximicen la función de densidad de probabilidad condicionada:

$$u = \arg\max_u p(u|r)$$

A partir del *teorema de Bayes* definimos la probabilidad condicionada:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)}$$

Definiendo las variables H como el modelo o hipótesis, y D como los datos u observaciones:

$$\begin{cases} A \equiv H \\ B \equiv D \end{cases} \qquad P(H|D) = \frac{P(D|H)P(H)}{P(D)}$$

Se tiene que $P(H|D)$ representa la distribución de probabilidad **a posteriori** de la hipótesis de los datos, $P(D|H)$ la función de **verosimilitud**, $P(H)$ la distribución de probabilidad **a priori** y $P(D)$ la probabilidad **marginal** de los datos.

Volviendo a la función a maximizar, seleccionaremos de entre todos los conjuntos posibles el que maximice la probabilidad a posteriori:

$$u = \arg\max_u p(u|r) = \arg\max_u p(u)p(r|u)$$

Como el término a maximizar es positivo podemos reformular la expresión anterior como:

$$u = \arg\max_u[\log p(u) + \log p(r|u)]$$

Y substituyendo,

$$u = \arg\min_u \left[-\log p(u) + \frac{1}{2\sigma^2}\int_\Omega \big(r(x) - u(x)\big)^2 dx\right]$$

El término $-\log p(u)$ es el "prior", es decir, la probabilidad a priori. En este caso utilizaremos como "prior" la variación total:

$$-\log p(u) = \mu \|u\|_{TV(\Omega)} = \int_{\Omega} |\nabla u| \, dx$$

Por lo que los conjuntos de valores que presenten un nivel mayor de variaciones serán candidatos más probables de ser la solución. Finalmente llegamos a la siguiente expresión que es equivalente a la propuesta en el método de *ROF*:

$$u = \arg\min_{u} \left[\mu \int_{\Omega} |\nabla u| \, dx + \frac{1}{2\sigma^2} \int_{\Omega} \big(r(x) - u(x)\big)^2 dx \right]$$

Vemos que la expresión toma la forma de la regularización de Tíjonov:

$$\min_{u} [F(u) \equiv \alpha V(u) + E(u)]$$

Donde $V(u)$ representa el **término regularizador** y $E(u)$ **el término de proximidad**.

1.1.2 Adaptación a diferentes modelos de ruido

La deducción de la ecuación de ROF haciendo uso de la estadística Bayesiana nos permite redefinir la ecuación para diferentes modelos de ruido.

- Ruido con distribución de Laplace:

$$u = \arg\min_{u} \left[\int_{\Omega} |\nabla u| \, dx + \lambda \int_{\Omega} |r(x) - u(x)| \, dx \right]$$

- Ruido de Poisson:

$$u = \arg\min_{u} \left[\int_{\Omega} |\nabla u| \, dx + \lambda \int_{\Omega} (u(x) - r(x) \log u(x)) \, dx \right]$$

1.2 MÉTODOS RESOLUTIVOS

Existen multitud de métodos resolutivos propuestos en el la literatura. El método que se introduce en la publicación de Rudin, Osher y Fatemi consiste en discretizar el algoritmo de gradiente descendente PDE. A partir de la siguiente expresión, obtenida en el apartado anterior:

$$u = \arg\min_{u} \left[F(u) \equiv \int_{\Omega} |\nabla u| \, dxdy + \lambda \int_{\Omega} \big(r(x) - u(x)\big)^2 dxdy \right]$$

En el proceso resolutivo se llegaría a la ecuación de Euler-Lagrange a resolver:

$$-\nabla \cdot \frac{\nabla \mathrm{u}(x)}{|\nabla \mathrm{u}(x)|} + \lambda\big(r(x) - u(x)\big)$$
$$\upsilon \cdot \nabla u(x) = 0 \;\; sobre \;\; \partial\Omega$$

Aplicando el método de gradiente descendiente se obtendrá la solución iterarando sobre la siguiente expresión discretizada (para los valores de dt y λ acudir a la publicación):

$$u_{i,j}^{n+1} = u_{i,j}^{n} + dt \left[\nabla_x^- \left(\frac{\nabla_x^+ u_{i,j}^n}{\sqrt{\left(\nabla_x^+ u_{i,j}^n\right)^2 + \left(m\left(\nabla_y^+ u_{i,j}^n, \nabla_y^- u_{i,j}^n\right)\right)^2}} \right) + \nabla_y^- \left(\frac{\nabla_y^+ u_{i,j}^n}{\sqrt{\left(\nabla_y^+ u_{i,j}^n\right)^2 + \left(m\left(\nabla_x^+ u_{i,j}^n, \nabla_x^- u_{i,j}^n\right)\right)^2}} \right) \right] + dt\lambda\left(r_{i,j} - u_{i,j}^n\right)$$

Capítulo 3. Sistemas microprocesadores

En este capítulo se realizará un breve análisis de las características técnicas de los sistemas microprocesadores más importantes a tener en cuenta en la implementación optimizada de algoritmos de visión artificial. Se mostrará la relación entre los diferentes aspectos técnicos y los métodos de codificación definiéndose los términos de localidad, redundancia y paralelización.

1. ASPECTOS TÉCNICOS A TENER EN CUENTA EN LA PROGRAMACIÓN

Se detallarán a continuación las características más relevantes de los sistemas microprocesadores que nos condicionarán el proceso de optimización.

1.1 PARALELISMO

1.1.1 A nivel de hilo

Un factor a tener en cuenta es el **número de procesadores** disponible y el número de núcleos de cada procesador. Esto nos determinará la capacidad del sistema para realizar cierto cálculo sobre un conjunto de datos de manera paralela, es decir, nos marcará el nivel de paralelización.

1.1.2 A nivel de datos

Un elemento muy habitual en el procesamiento de señales es el filtrado. Un filtro básico realiza una operación tal que:

$$out[n] = a_0 in[n] + a_1 in[n-1] + a_2 in[x-2] + a_3 in[x-3]$$

En un procesador con arquitectura *SISD* (*Single Instruction, Single Data*), se deberían cargar en el procesador cuatro instrucciones de multiplicación que operasen con sus respectivos datos. Sin embargo actualmente la mayoría de microprocesadores de uso habitual implementan una **arquitectura *SIMD*** (*Single Instruction, Multiple Data*). Esta arquitectura permite ejecutar con una sola instrucción la misma operación sobre un número determinado de datos. En programación, a la técnica de ejecutar una misma operación sobre un conjunto de datos se le denomina vectorización, esto es que se aplicará una operación a cada elemento de un vector. Volviendo al ejemplo del filtro indicado, con una implementación vectorizada se reduciría el número de ciclos necesarios para obtener el resultado con respecto a una ejecución secuencial.

1.2 MEMORIA VOLÁTIL

Otro aspecto importante es el relativo a la memoria. Para ello hay que analizar las prestaciones de cada tipo de memoria del sistema microprocesador. En la figura 3.1 se muestra un diagrama de la jerarquía de memoria comparando las propiedades de capacidad y velocidad de acceso de los diferentes tipos de memoria.

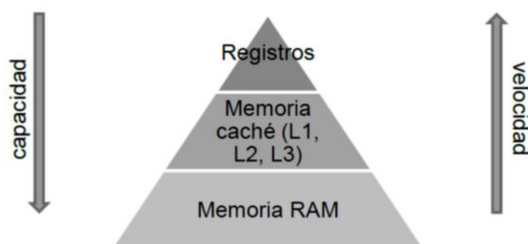


Figura 3.1: Niveles de memoria

El **tamaño de la memoria caché** nos determinará, por ejemplo, el tamaño óptimo de los búferes que almacenen cálculos temporales. Es decir, si para la realización de una serie de cálculos hay que utilizar de manera reiterada otros valores calculados previamente, nos conviene tener guardados estos valores temporales lo más próximos posible al núcleo. Si el tamaño disponible de este tipo de memoria es muy reducido los cálculos temporales se almacenarán en memoria RAM. En este caso hay que considerar la **velocidad de lectura/escritura de la memoria primaria** (RAM), ya que si la velocidad de acceso a esta es muy lento hay que analizar otras opciones de programación como puede ser el cómputo redundante de datos. En la codificación, el parámetro que nos indicará el aprovechamiento de los niveles de memoria más próximos al núcleo es la localidad de los cálculos, es decir, el tiempo que transcurre (*distancia*) desde que se computa un valor hasta que este es utilizado.

1.3 INSTRUCCIONES POR SEGUNDO (IPS)

Este parámetro nos informará acerca de la capacidad de cómputo de cada núcleo del procesador, dándonos una idea acerca del coste de evaluar una operación. Si este coste muy elevado habrá que reducir la redundancia en el cálculo.

2. RELACIÓN ENTRE LAS PRESTACIONES DEL SISTEMA EN EL PROCESADO DE IMÁGENES

Se han introducido en el apartado anterior los conceptos de **paralelización**, **localidad** y **redundancia**. Estos parámetros están influidos por las características del sistema microprocesador sobre el que se trabaje y, a su vez, estarán relacionados unos con otros. Para ilustrar como se influyen mutuamente estos parámetros partiremos de la implementación de ciertos casos extremos a partir de la siguiente expresión, donde a la función $g(x,y)$ la llamaremos función productora (*productor*) – la que generará los valores necesarios para un cálculo posterior – y a la función $f(x,y)$ función consumidora (*consumidor*) – la que utilizará los valores generados por otra función.

$$g(x,y) = \log|x+y|\,;$$
$$f(x,y) = g(x,y) + g(x+1,y) + g(x,y+1) + g(x+1,y+1)$$

2.1 ESQUEMA DE CÁLCULO EN LA RAÍZ: SOBRECARGA DE LA MEMORIA PRIMARIA

Para obtener la menor redundancia posible, una vía para evaluar los puntos de la función consumidora sería evaluar previamente la función productora en su totalidad y almacenar los valores temporalmente. De este modo solo estamos computando una única vez cada punto de la función $g(x,y)$ como observamos en la figura 3.2.

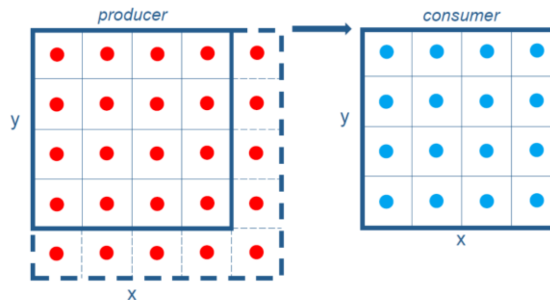


Figura 3.2: Esquema de procesamiento en la raíz

Sin embargo en este caso extremo el grado de paralelización es nulo y la localidad de los valores temporales es muy pobre ya que la distancia entre el cálculo de los valores de la función productora hasta su utilización en la evaluación de los puntos de la función consumidora es muy grande. Que la localidad sea tan pobre nos está indicando un uso muy deficiente de la memoria volátil, y la inexistencia de paralelización nos indica que no se está haciendo un uso intensivo del núcleo.

Podemos ilustrar este modelo de cálculo tal como se muestra en la figura 3.3, donde los vértices del triángulo indican el nivel óptimo de cada parámetro. En este caso tenemos el mejor nivel de redundancia (es decir, nulo), sin embargo este esquema presenta un grado deficiente en localidad y paralelismo.

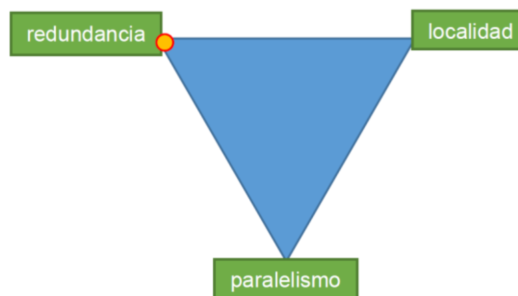


Figura 3.3: Propiedades del procesamiento en la raíz

Sistemas microprocesadores objetivo: Si se dispone de un sistema microprocesador con un acceso rápido a la memoria primaria y solo existe un único núcleo con una velocidad de cómputo muy lenta, quizás esta sea un buen esquema de implementación optimizada ya que se hace uso intensivo del acceso a memoria RAM y se utiliza de manera reducida el único núcleo existente en el sistema.

2.2 Esquema de cálculo insertado: Sobrecarga del núcleo

Otro esquema posible en el que la localidad sería máxima es el cálculo de los puntos de la función productora según se vayan necesitando en la función consumidora.

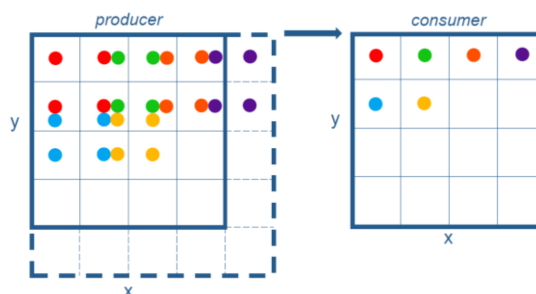


Figura 3.4: Esquema de procesamiento insertado

Este esquema presenta el peor nivel de redundancia posible para la implementación ya que, como se observa en la figura 3.4, la mayoría de los puntos de evaluarán cuatro veces. De nuevo, la paralelización es inexistente.

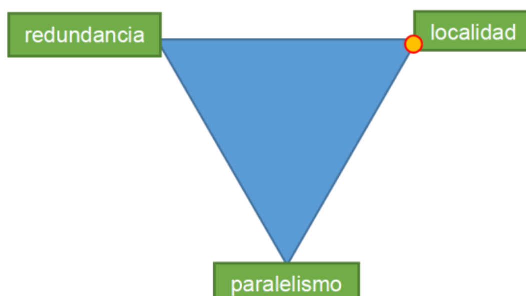


Figura 3.5: Propiedades del procesamiento insertado

En este caso extremo nos encontramos con una situación opuesta al de cálculo en la raíz (figura 3.5). Es decir, en esta ocasión no se hace uso de la memoria volátil del sistema (buena localidad), por el contrario, se está sobrecargando el nivel de cómputo en el núcleo (mucha redundancia).

Sistemas microprocesadores objetivo: Por tanto, si disponemos de un procesador de elevadas prestaciones en la capacidad de cómputo pero que dispones de un acceso a memoria primaria lento y que dispone de memoria caché reducida, puede ser un buen esquema de programación óptima aligerar el uso de memoria volátil a costa de sobredimensionar el cómputo (aumentar la localidad a costa de incrementar la redundancia).

2.3 ESQUEMA DE CÁLCULO PARALELIZADO: OPCIÓN DE COMPROMISO

En este esquema aplicaremos cálculo concurrente. De esta manera se evaluará la función consumidora por procesos independientes, no obstante también hay que elegir el esquema de cálculo de cada proceso (*hilo*). En este ejemplo se aplicará el esquema de cálculo en la raíz de manera paralelizada, mostrando así la dependencia de los parámetros de localidad, redundancia y paralelismo.

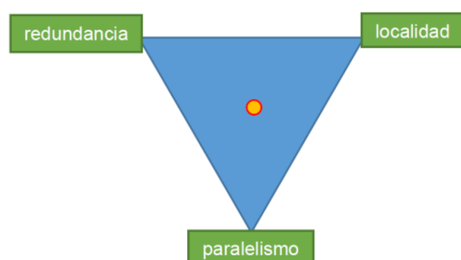


Figura 3.6: Propiedades del procesamiento paralelizado

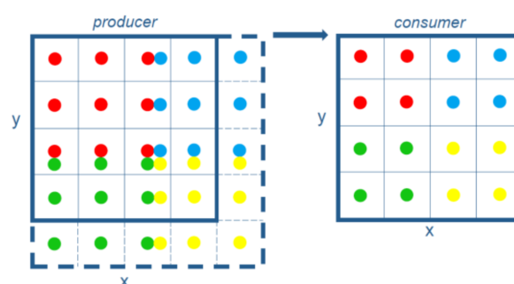


Figura 3.7: Esquema de paralilzado

Como se puede apreciar en la figura 3.7, partiendo del cálculo en la raíz se ha paralelizado la evaluación de la función consumidora. Esto nos produce una reducción de la localidad, es decir, los valores temporales se utilizarán más rápidamente en la evaluación de los puntos de la función consumidora. De esta manera se ha mejorado el uso de la memoria volátil, ya que si las regiones son del tamaño adecuado, pueden almacenarse en la memoria caché. Sin embargo se ha generado un incremento de la redundancia, ya que ciertos puntos de la función productora se utilizarán en dos o más procesos independientes, por lo que cada proceso deberá evaluarlos de manera individual. El uso de procesado concurrente explotará las prestaciones de los procesadores multinúcleo.

Por lo tanto, este esquema corresponde a una solución de compromiso entre los tres parámetros fundamentales a considerar en la implementación optimizada de algoritmos de procesado de imagen (figura 3.6).

Capítulo 4. Actuales herramientas de codificación

En este capítulo se analizan las diferentes herramientas y técnicas de programación disponibles en la actualidad para la implementación de algoritmos de procesado de imagen en lenguaje C++. Se presentaran las librerías específicas en el área de visión artificial más utilizadas, así como las librerías de álgebra lineal que resultan útiles en la implementación de este tipo de algoritmos. Finalmente se introducen otras técnicas de programación avanzada.

1. LIBRERÍAS DE VISIÓN ARTIFICIAL

Existen diversas librerías de visión artificial disponibles en la actualidad, aunque quizás la más popular y extendida sea **OpenCV** [14]. La librería *Open Source Computer Vision* es una librería desarrollada en C++ que implementa multitud de funciones para su uso en aplicaciones de visión artificial en tiempo real y en aplicaciones de aprendizaje automático (*machine learning*). Cuenta con cerca de 2500 algoritmos implementados de manera optimizada, y muchas más son las funciones que implementa para dar soporte a estos algoritmos. En sus orígenes fue desarrollada por un grupo de investigación de Intel en Rusia, pero en la actualidad es la comunidad la que continua con su desarrollo. Es una librería muy madura y cuenta con mucha documentación en la red. Se está estudiando desde *Khronos Group* su estandarización como API para visión artificial.

A partir de la descarga de su código fuente podemos compilar la librería para multitud de sistemas operativos (Windows, Linux, Android/Linux, iOS,…) y arquitecturas de procesador (x86-64, ARMv7 y ARMv8). Cuenta también con interfaces para Python, Java y Matlab que nos permiten desarrollar aplicaciones en cada plataforma. En el proceso de compilación se recomienda hacer uso del programa CMake, en el que de una manera gráfica podemos seleccionar las opciones de compilación. Entre estas opciones hay que prestar atención a las relativas a la optimización (TBB, IPP, Eigen, SIMD adecuadas al procesador – disponibilidad de SSSE3 y SSE 4.2 –, OpenCL, uso de GPU – CUDA –, etc.).

Otra librería disponible es **VXL** (the *Vision-something-Libraries*), que se define a sí misma como una librería para la implementación e investigación en el campo de la visión artificial. Esta librería está compuesta de una serie de núcleos que dan un soporte más amplio, es decir, no es una librería que únicamente implemente las funciones habitualmente utilizadas en visión artificial. Estos núcleos son: *vnl* (cálculo numérico), *vil* (manipulación de imágenes), *vgl* (geometría), *vsl* (flujos de entrada y salids), *vul* (utilidades), etc.

También encontramos librerías desarrolladas por los fabricantes de hardware que implementan rutinas de bajo nivel para la codificación de aplicaciones de imagen y video haciendo uso de las características de sus respectivos dispositivos. Librerías de este tipo son:

IPP de Intel, **CoreImage** de Apple, **NPP** de NVIDIA, **VLIB** e **IMGLIB** de Texas Instruments, **FastCV** de Qualcomm, etc.

Cabe destacar FastCV, que si bien no incorpora la multitud de funciones disponibles en OpenCV, se trata de una librería optimizada para el uso en dispositivos móviles (sistemas embebidos). Se dispone de una API bien documentada y existen multitud de ejemplos. Ambas librerías, así como VXL, son de uso gratuito y pueden distribuirse junto a aplicaciones comerciales, es decir, se puede desarrollar software comercial haciendo uso de estas librerías.

Existen librerías comerciales como las de **Visage Technologies**. Estas librerías son software privativo por lo que hay que contar con una licencia para su uso. No obstante cuentan con una API muy bien desarrollada y bien documentada, además de ofrecer soporte para multitud de plataformas (Linux, Windows, Android/Linux, iOS, Unity, FPGA, etc.).

2. LIBRERÍAS DE ÁLGEBRA LINEAL

También puede resultar útil la utilización de librerías de cálculo lineal en la implementación de algoritmos de visión artificial. Librerías como OpenCV nos ofrecen multitud de funciones ya implementadas que pueden ser útiles en la implementación de proyectos de visión más complejos, sin embargo (como puede ser en la implementación de un determinado filtro) puede darse el caso que no necesitemos utilizar funciones específicas de visión y necesitemos librerías específicas para el cálculo lineal que estén más optimizadas. No obstante cabe destacar la interrelación existente entre la mayoría de estas librerías, es decir, muchas librerías hacen a su vez uso de otras librerías. Por ejemplo, OpenCV hace uso de la librería de álgebra lineal Eigen. La librería de álgebra lineal Armadillo también hace uso de otras librerías matemáticas como LAPACK, ATLAS, MKL o ACML para determinadas operaciones.

Eigen [5] es una librería de plantillas en C++ para álgebra lineal (cálculo matricial, vectorial, análisis numérico,…). Su implementación está muy optimizada y paraleliza a nivel de datos mediante vectorización con instrucciones SIMD.

Automatically Tuned Linear Algebra Software (ATLAS) [2] es una librería para álgebra lineal. Es una implementación del estándar BLAS. Basic Linear Algebra Subprograms (BLAS) es una especificación de rutinas de bajo nivel para operaciones de álgebra lineal. ATLAS proporciona una implementación optimizada del estándar BLAS según las características de la máquina en la que se va a utilizar. También proporciona funciones adicionales a través de la librería LAPACK.

LAPACK es una librería que ofrece soporte para la resolución de sistemas lineales de ecuaciones, mínimos cuadrados, descomposición en valores propios y descomposición en valores singulares. Se basa en el estándar BLAS.

OpenBLAS [13] es otra librería que proporciona una implementación del estándar BLAS optimizada de manera automática según el tipo de máquina.

Armadillo [1] es una librería en C++ para álgebra lineal. Es muy útil para una rápida implementación en fase de producción de algoritmos desarrollados en la etapa de investigación ya que cuenta con una API muy similar a Matlab. Puede ser utilizada para diferentes áreas de trabajo como aprendizaje automático, reconocimiento de patrones, procesado de la señal, etc. Diversas descomposiciones matriciales utilizan la librería LAPACK, pudiendo ser reemplazada por otras similares.

GNU Scientific Library (GSL) es una librería matemática de propósito general desarrollada en C para el cálculo numérico en aplicaciones matemáticas y científicas. Es una librería que cubre muchas áreas como generadores de números aleatorios, ecuaciones diferenciales, interpolaciones, aproximaciones de Chebyshev y muchas otras.

2.1 EVALUACIÓN DE PRESTACIONES

En [9] encontramos una evaluación del rendimiento de las diferentes librerías expuestas anteriormente. La batería de pruebas consiste en la resolución de un sistema lineal de ecuaciones de la forma:

$$AX = B$$

Resolviéndose mediante el siguiente *algoritmo*:

$$X = (A^T A)^{-1} A^T B$$

Se estudia el código ofrecido para la batería de pruebas, así como los métodos de medida implementados. Se concluye que la implementación ofrecida es adecuada y se ejecuta en una máquina con procesador Intel Core i7 3610QM a 2'3GHZ, validándose los resultados ofrecidos en la referencia.

Para unas dimensiones de A de 1.000.000x16 y B de 1.000.000x1 se obtienen los resultados mostrados en la figura 4.1.

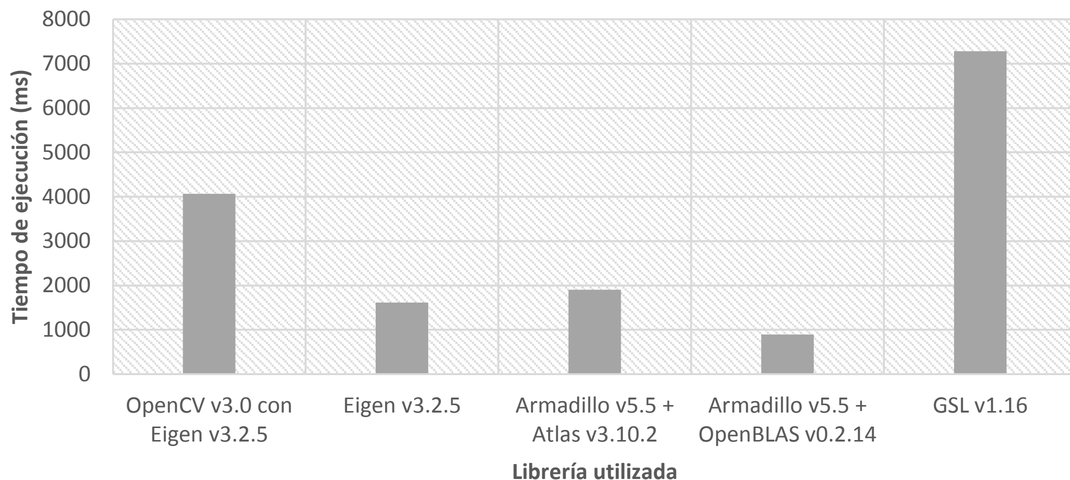


Figura 4.1: Evaluación del rendimiento de la diferentes librerías disponibles

3. OTRAS HERRAMIENTAS PARA UNA IMPLEMENTACIÓN OPTIMIZADA

Existen también otras muchas librerías que ayudan a la implementación de código optimizado. Una de ellas es la **biblioteca Boost**, que es un conjunto de librerías para el desarrollo de código avanzado en C++. Encontramos en ella librerías que aportan nuevas funcionalidades al lenguaje en áreas como la metaprogramación en plantillas, programación de algoritmos, procesado de imágenes, programación concurrente, estructurasm etc. Para el soporte en la programación concurrente tenemos la librería de plantillas **Threading Building Blocks** (*TBB*), la cual es utilizada por OpenCV como se comentó anteriormente. En este sentido también es muy utilizado **OpenMP**, que a través de una API permite la programación concurrente siguiendo un modelo *fork-join*.

También hay una gran cantidad de software disponible enfocado a la producción de código optiminazado. **OpenTuner** es un *framework* de programación que ayuda a la optimización paramétrica de código. Un ejemplo de este tipo de optimización es el que encontramos en la librería ATLAS, el cual adapta diferentes parámetros en el código para obtener el mejor rendimiento de acuerdo a las características de la máquina en la que se está compilando. OpenTuner también es utilizado por el lenguaje de programación *HALIDE* (el cuál se explica en apartados posteriores) para obtener el mejor esquema de codificación del algoritmo.

Capítulo 5. Lenguaje de programación específico HALIDE

HALIDE es un nuevo lenguaje de programación desarrollado para facilitar la implementación de algoritmos de procesado de imagen de una manera eficiente. En este capítulo se presenta este lenguaje de programación explicando de manera resumida su estructura interna así como sus primitivas, esquemas de optimización, modos de compilación, etc.

1. LENGUAJE EMBEBIDO EN C++

HALIDE es un lenguaje específico del domino embebido en C++. Un lenguaje específico del dominio (del inglés *domain-specific language – DSL*) es un tipo de lenguaje especializado para el desarrollo de aplicaciones en un área del conocimiento determinada. Ejemplos de lenguajes específicos del dominio son HTML, Verilog y VHDL, Matlab, SQL, etc. Por el lado contrario encontraríamos los lenguajes de programación de propósito general como C o Java.

Para poder programar utilizando el lenguaje específico *HALIDE* no debemos instalar ningún tipo de programa ni hacer uso de ningún compilador especial. *HALIDE* se ha desarrollado embebido en C++ a través de una librería compartida y se compila haciendo uso de las herramientas habituales como GCC. Esto no es del todo cierto porque realmente existe un proceso de compilación propio de *HALIDE* pero que se ejecuta dentro del código compilado. Es decir, un programa habitual codificado en C++ se escribirá utilizando una sintaxis determinada. Cuando se inicia el proceso de compilación se llamará a un ejecutable (compilador) y se le pasará como entrada este código fuente, que es un fichero de texto, pasando al *front-end* del compilador. *El front-end* realizará las tareas de verificar la sintaxis y la semántica (generar advertencias y errores) y generará una representación intermedia (en inglés *intermediate representation* o *IR*). A partir de esta representación intermedia se generará el código máquina mediante las etapas posteriores. Sin embargo, el *front-end* de *HALIDE* no cuenta con un fichero de texto como entrada, sino que directamente se le pasan una serie de estructuras que contaran con una serie de tipos y operadores/funciones que se encuentran definidas en el espacio de nombres (*namespace*) de *HALIDE*. Durante el proceso de compilación de *HALIDE* se optimizará la representación intermedia y finalmente esta se mapeará directamente a la representación intermedia de LLVM (aunque partes de la representación intermedia de *HALIDE* serán mapeadas directamente a códigos de operación, en inglés *opcodes*, SIMD propios de la arquitectura con la que se está trabajando). Todo este proceso de compilación de *HALIDE* se produce en el momento de la ejecución del programa compilado en C++ utilizando GCC haciendo uso de la librería de *HALIDE*, es decir, el compilador de *HALIDE* es el propio programa generado. Una vez generado el resultado de la

compilación, este código máquina generado puede ser enviado bien a un fichero en disco (generar el fichero objeto con un formato como *Executable and Linkable Format – ELF*), o bien directamente a la memoria volátil en tiempo de ejecución. En este sentido el compilador de *HALIDE* puede comportarse de dos maneras diferentes según donde guarde el código máquina generado:

- **Ahead-of-time compiler** (*AOT*). Este es el funcionamiento de compiladores como GCC para lenguaje C o *javac* para lenguaje *Java*. Se obtiene como salida un fichero objeto (*object file*) con el formato adecuado el cual incorpora el algoritmo implementado como una función compatible con la interfaz binaria de aplicaciones de C (*C ABI-compatible*), es decir, que esta función puede ser llamada desde otros programas codificados en lenguaje C.
- **Just-in-time compiler** (*JIT*). Se ejecuta el programa compilado con GCC en el que se ha definido en alguna parte de este la estructura de *HALIDE*. En tiempo de ejecución se generará el código máquina necesario para ejecutar el algoritmo definido con *HALIDE* y este código se enviará directamente a la memoria volátil listo para ser ejecutado.

2. ESTRUCTURA BÁSICA DEL PROGRAMA

Como se explicó en la introducción, la etapa más laboriosa en el proceso de integración de un algoritmo de visión artificial en un sistema microprocesador es la etapa de optimización y adaptación a la arquitectura y las prestaciones del sistema procesador objetivo. El lenguaje de programación *HALIDE* se basa en la separación de dos conceptos clave:

- **Algoritmo**. Es la definición matemática del proceso. Esta definición será igual para todas las optimizaciones que se generen para cada caso determinado. Es la base del trabajo realizado en la investigación y representa la solución al problema. En *HALIDE* esta parte no se encarga en absoluto de cómo se van a computar los datos, sino del resultado que deseamos tener a la salida de nuestra implementación. La definición que se haga en esta sección se respetará y el proceso de optimización nunca afectará al resultado obtenido.
- **Esquema**. En esta parte se definirá como se computará el algoritmo anteriormente definido.

Es decir, *HALIDE* nos permite separar el *qué* se calcula del *cuando* se computa, *dónde* se guardan los datos y *cómo* se procesa la imagen. La ventaja de tener separada la definición del algoritmo de la definición del esquema de procesamiento nos permitirá probar una gran cantidad de esquemas y seleccionar el más óptimo para cada caso. Además, la facilidad y simplicidad en la programación nos permitirá una rápida portabilidad a otras arquitecturas y sistemas sin afectar a la definición del algoritmo en sí, ya que la salida siempre será la definida en la primera sección.

3. DEFINICIÓN DEL ALGORITMO

Como se ha explicado esta sección tiene como finalidad la de definir el comportamiento del algoritmo. Se partirá de una expresión del tipo:

$$r(x,y) = \nabla_x^+ u(x,y)$$

La cual se codificará siguiendo la sintaxis de *HALIDE*:

$$r(x,y) = input(x+1,y) - input(x,y)$$

3.1 PRIMITIVAS DEL LENGUAJE

Las primitivas básicas del lenguaje son las siguientes:

- **Función**. Representa una definición de los valores de una imagen evaluados en un dominio determinado. Una función tendrá como argumento un número determinado de variables que representaran la dimensionalidad de la función. Para facilitar la depuración podemos asignar un nombre en la declaración de la función para que al generarse avisos o errores en la compilación nos resulte más fácil la localización de la misma.

```
Halide::Func r("resultado");
r(x, y, c) = input(x+1, y, c) - input(x+1, y, c);
```

- **Variable**. Representa una dimensión en el dominio de la función.

```
Halide::Var x("x"), y("y"), c("c");
```

- **Dominio reducido**. Representa cierto dominio sobre el que se evaluaran ciertas funciones.

```
Halide::Rdom s(0, 50);
f(x, s) = f(x, s) * f(x, s);
```

- **Expresión**. Una expresión define el valor de una función en términos de las variables y otras funciones.

```
Halide::Expr e = input(x+1, y, c) - input(x+1, y, c);
r(x, y, c) = e;
```

- **Imagen**. Es una referencia inmutable a un buffer externo de memoria, visible para el algoritmo como una función definida únicamente sobre el dominio de la imagen (sus dimensiones).

```
Halide::Image<uint8_t> input = load<uint8_t>("test.png");
```

- **Parámetro**. Es una variable que se utiliza para definir determinados valores utilizados en la ejecución del algoritmo.

```
Halide::Param<uint8_t> offset;
r(x, y, c) = r(x, y, c) + offset;
```

4. DEFINICIÓN DEL ESQUEMA

Una vez definido el algoritmo en la sección anterior, y comprobado su resultado, debemos definir el esquema del programa. Esto comprende dos partes:

- El orden en que se evalúan los píxeles de la función a calcular.
- El orden en que se evalúan y se almacenan los valores de las etapas anteriores.

4.1 ESQUEMAS DE EVALUACIÓN DE UNA FUNCIÓN

Empezaremos por exponer los diferentes métodos de evaluación de una función. En este apartado se utilizará código C simplificado para que visualmente se comprenda mejor el esquema del programa, por lo que no se respetará la sintaxis al completo. Partiremos de la siguiente definición del algoritmo.

$$result(x, y) = x * y;$$

4.1.1 Orden por defecto

Por defecto se evaluarán los píxeles por filas. Representando la variable x las columnas y la variable y las filas, tenemos que la variable x se iterará variando esta rápidamente mientras que la variable y se iterará variando lentamente.

```
result.realize(5, 5);
```

En este caso no se ha indicado nada acerca de la esquema y se está llamando a la realización de la función en tiempo de ejecución (se detallará más adelante la manera de llamar a los dos métodos de compilación). Este esquema tiene un código C equivalente:

```
for( y = 0; y < 5; y++ )
    for( x = 0; x < 5; x++ )
        result[y][x] = x * y;
```

De manera gráfica tenemos que el orden de evaluación sería el mostrado en la figura 5.1.

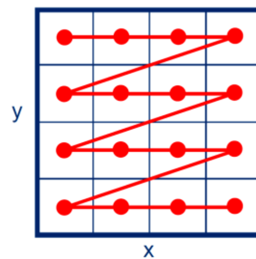


Figura 5.1: Esquema de procesado por defecto

4.1.2 Reordenación de variables

Se redefine el orden en el que se iteran las variables de la función. Se indica el orden de la variables en el argumento del método, siendo el primer argumento la variable más interna (la que variará más rápidamente).

```
derivative.reorder(y, x);
result.realize(4, 4);
```

De esta manera estaremos evaluando los píxeles de la función por columnas, con el siguiente esquema en código C equivalente:

```
for( x = 0; x < 4; x++ )
    for( y = 0; y < 4; y++ )
        result [y][x] = x * y;
```

De manera gráfica tenemos que el orden de evaluación sería el mostrado en la figura 5.2.

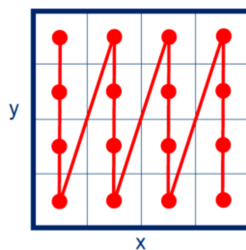


Figura 5.2: Esquema con reordenación de variables

4.1.3 División de variables

La división de una variable en dos sub-variables nos resultará muy útil en combinación con otras técnicas, aunque por sí sola no cambia el orden de evaluación de los píxeles.

```
result.split(x, x_outer, x_inner, 2);
result.realize(4, 4);
```

El primer argumento del método es la variable que se desea dividir. La siguientes dos variables son la variable sobre la que se iterará en el bucle externo y la variable sobre la que se iterará en el bucle interno, respectivamente. El último parámetro corresponde al factor de división. La variable del bucle interno iterará desde cero hasta el factor de división, mientras que la variable del bucle externo iterará desde cero hasta el valor previo dividido por el factor de división.

```
for( y = 0; y < 4; y++ )
    for( x_outer = 0; x_outer < 2; x_outer++ )
        for( x_inner = 0; x_inner < 2; x_inner++ )
            x = x_outer * 2 + x_inner;
            result[y][x] = x * y;
```

Si el factor de división no resulta en un cociente exacto con la variable a dividir se producirá un pequeño solape, de número de píxeles igual al valor del resto, en la evaluación de la última muestra. Esto solo sucederá en algoritmos en los que no importe evaluar dos veces el mismo píxel, si no es el caso se redefinirá el dominio de la función automáticamente para no variar el resultado del algoritmo.

```
for( y = 0; y < 4; y++ )
    for( x_outer = 0; x_outer < 3; x_outer++ )
        for( x_inner = 0; x_inner < 2; x_inner++ )
            x = x_outer * 2;
            if ( x > 3 )
                x = 3;
            x += x_inner;
            result[y][x] = x * y;
```

4.1.4 Fusión de variables

Consiste en fusionar las variables de dos bucles en una sola variable sobre la que iterar.

```
result.fuse(x, y, fused);
result.realize(4, 4);
```

Los dos primeros argumentos corresponden a las variables a fusionar y el último a la variable sobre la que se iterará. Dentro de la iteración de esta variable se obtendrán los correspondientes a las variables originales. Esta técnica no varía el orden de evaluación de los píxeles.

```
for( fused = 0; fused < 4*4; fused++ )
    x = fused / 4;
    y = fused % 4;
    result[y][x] = x * y;
```

4.1.5 Evaluación en mosaico

Del inglés *tiles*, resulta un término difícil de traducir con una palabra quizás más técnica. La traducción literal sería *evaluación en baldosas*, lo que se corresponde muy gráficamente con el resultado de esta técnica. El resultado es la evaluación de la imagen completa a través de subregiones cuadradas de un tamaño determinado (las *baldosas*). Para conseguir este efecto

dividiremos las dos variables (filas y columnas) por un mismo factor y reordenaremos su evaluación.

```
result.split(x, x_outer, x_inner, 2);
result.split(y, y_outer, y_inner, 2);
result.reorder(x_inner, y_inner, x_outer, y_outer);
result.realize(4, 4);
```

O haciendo uso directamente del método siguiente:

```
result.tile(x, y, x_outer, y_outer, x_inner, y_inner, 2, 2);
```

Este esquema tiene un código C equivalente:

```
for( y_outer = 0; y_outer < 2; y_outer++ )
        for( x_outer = 0; x_outer < 2; x_outer ++ )
                for( y_inner = 0; y_inner < 2; y_inner++ )
                        for( x_inner = 0; x_inner < 2; x_inner++ )
                                x = x_outer * 2 + x_inner;
                                y = x_outer * 2 + y_inner;
                                result[y][x] = x * y;
```

De manera gráfica tenemos que el orden de evaluación sería el mostrado en la figura 5.3.

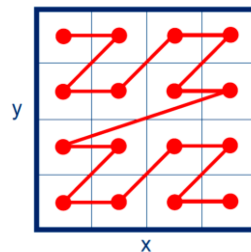


Figura 5.3: Esquema de procesamiento en mosaico

4.1.6 Evaluación en vectores

Esta técnica se basa en explotar las arquitecturas que soportan paralelismo a nivel de datos, esto es que para una sola instrucción se realizará la misma operación sobre un conjunto de datos de manera simultánea (vector de datos). Este concepto se explica en el apartado de sistemas microprocesadores.

```
result.split(x, x_outer, x_inner, 4);
result.vectorize(x_inner);
result.realize(8, 4);
```

Consiste en crear un bucle interno que itere desde cero hasta el nivel de vectorización adecuado a la arquitectura y, de manera interna a través de SIMD, vectorizar la evaluación del bucle. También puede vectorizarse de manera directa:

```
result.vectorize(x, 4);
result.realize(8, 4);
```

En el siguiente código no se implementan estas instrucciones ya que la intención es visualizar el efecto de cada técnica para facilitar su entendimiento, por lo que considérese que la evaluación de los píxeles se realiza de manera vectorizada (paralela).

```
for( y = 0; y < 5; y++ )
        for( x_outer = 0; x_outer < 2; x++ )
                result[y][x_outer*4+0] = ( x_outer*4 + 0 ) * y;
```

```
                    result[y][x_outer*4+1] = ( x_outer*4 + 1 ) * y;
                    result[y][x_outer*4+2] = ( x_outer*4 + 2 ) * y;
                    result[y][x_outer*4+3] = ( x_outer*4 + 3 ) * y;
```

4.1.7 Desenrollar un bucle

En ocasiones puede que un conjunto de píxeles compartan ciertos datos en su evaluación. En estos casos una técnica de optimización es la de desenrollar un bucle y así computar o cargar desde memoria el dato compartido una única vez.

```
result.split(x, x_outer, x_inner, 2);
result.unroll(x_inner);
result.realize(4, 4);
```

Este esquema tiene un código C equivalente:

```
for( y = 0; y < 5; y++ )
      for( x_outer = 0; x_outer < 2; x++ )
            x_inner = 0;
            x = x_outer*2 + x_inner;
            result[y][x] = x * y;

            x_inner = 1;
            x = x_outer*2 + x_inner;
            result[y][x] = x * y;
```

4.1.8 Paralelizado

Se mostrará el uso de la técnica de paralelizado a través de la división de la imagen en cuatro subregiones que se evaluarán de manera paralela.

```
result.tile(x, y, x_outer, y_outer, x_inner, y_inner, 2, 2);
result.fuse(x_outer, y_outer, tile_index);
result.parallel(tile_index);
result.realize(4, 4);
```

A partir de la primera línea dividimos las variables en 2 para generar el efecto mosaico en la imagen. Seguidamente fusionamos las subvariables de cada bucle externo respectivo para que cada evaluación de la variable fusionada corresponda a una región de 2x2 píxeles. Esta variable (`tile_index`) es la que paralelizaremos, aunque por cuestiones de simplificación en el código C equivalente se ha expresado como la evaluación de la variable en un bucle.

```
for( tile_index = 0; tile_index < 4; tile_index ++ )
      x_outer = tile_index / 2;
      y_outer = tile_index % 2;
      for( y_inner = 0; y_inner < 2; y_inner++ )
            for( x_inner = 0; x_inner < 2; x_inner++ )
                  y = x_outer * 2 + y_inner;
                  x = x_outer * 2 + x_inner;
                  result[y][x] = x * y;
```

De manera gráfica tenemos que el orden de evaluación sería el mostrado en la figura 5.4. En esta figura se observan las iteraciones de `tile_index` en diferentes colores, esto es, cada color del grafo corresponde a un hilo en ejecución.

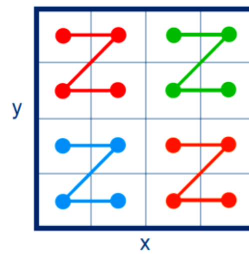


Figura 5.4: Esquema de procesamiento en paralelo

4.2 ESQUEMA DE CÁLCULO MULTI-ETAPAS

El proceso de optimización de código se basa en encontrar un compromiso entre el **paralelismo** (evaluación de diferentes regiones de la imagen de manera simultánea), la **localidad** de los datos (el tiempo que transcurre desde que se calcula un valor hasta que se utiliza) y el **cálculo redundante** (evaluación de un mismo píxel más de una vez), tal y como se analiza en el apartado de sistemas microprocesadores. Para mostrar el uso de *HALIDE* para diferentes técnicas utilizaremos el siguiente algoritmo:

$$g(x, y) = \log|x + y| ;$$
$$f(x,y) = g(x,y) + g(x + 1, y) + g(x, y + 1) + g(x + 1, y + 1)$$

A la función $g(x,y)$ la llamaremos función productora (**productor**) y la función $f(x,y)$ será la que utilice los datos de esta función, es decir la función consumidora (**consumidor**).

```
producer(x, y) = log( x + y );
consumer(x, y) = producer(x, y) + producer(x+1, y) + producer(x, y+1) +
      producer(x+1, y+1);
```

4.2.1 Orden por defecto

Cuando no se especifica nada acerca del orden de evaluación de la función productora se compilará de manera **insertada** (en inglés el término es *inline*). Esto significa que en la función consumidora se substituirán las llamadas a la función productora por su expresión correspondiente (se *insertará* la expresión en las llamadas correspondientes).

```
consumer.realize(4, 4);
```

Es decir, para cada píxel de la función consumidora se evaluarán los píxeles necesarios de la función consumidora según se vayan necesitando. Esto provocará un gran nivel de redundancia sobre la evaluación de los píxeles de la función productora, sin embargo presenta un buen nivel de localidad ya que los píxeles evaluados en la función productora serán utilizados de manera inmediata en la evaluación de la función consumidora.

```
for( y = 0; y < 4; y++ )
      for( x = 0; x < 4; x++ )
            consumer[y][x] = log(x + y) + log((x+1) + y) + log(x + (y+1))
                  +  log((x+1) + (y+1));
```

De manera gráfica obsérvese en la figura 5.5 como por cada píxel evaluado en la función consumidora se están evaluando cuatro píxeles en la función productora. Nótese el alto grado de localidad y el alto grado de redundancia. Para este caso en concreto todos los píxeles de la función productora, excepto los correspondientes a los bordes de la imagen, serán evaluados cuatro veces.

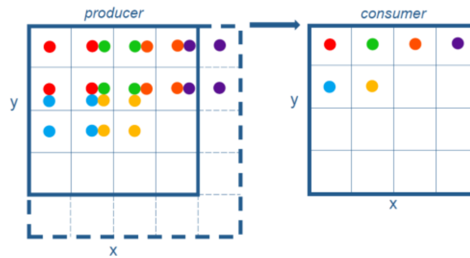


Figura 5.5: Esquema de evaluación de la función productora por defecto

4.2.2 Evaluación en el inicio

Se trata de evaluar la función productora en su totalidad antes de ser utilizados sus datos por la función consumidora. Es decir, se evaluarán los píxeles de la función productora y se guardarán en la memoria volátil, desde donde se cargarán a medida que se necesiten en la evaluación de los píxeles de la función consumidora. Por lo tanto, el nivel de redundancia será mínimo (los píxeles de la función productora se evaluarán una única vez) pero presenta un bajo nivel de localidad (la distancia entre el cálculo de un pixel de la función productora y su uso en la función consumidora es elevada, por lo que no se optimiza el cálculo a través del aprovechamiento de la memoria caché del procesador).

```
producer.compute_root();
consumer.realize(4, 4);
```

Este procedimiento equivale a evaluar las funciones de manera separada:

```
producer_buffer[5][5];
for( y = 0; y < 5; y++ )
       for( x = 0; x < 5; x++ )
              producer_buffer[y][x] = log(x + y);
for( y = 0; y < 4; y++ )
       for( x = 0; x < 4; x++ )
              consumer[y][x] = producer_buffer[y][x]
                     + producer_buffer[y][x+1]
                     + producer_buffer[y+1][x]
                     + producer_buffer[y+1][x+1];
```

4.2.3 Evaluación por filas

Antes de empezar a evaluar los píxeles de una determinada fila de la función consumidora, se evaluarán todos los píxeles de la función productora que se necesitarán para esa determinada fila.

```
producer.compute_at(consumer, y);
consumer.realize(4, 4);
```

Para el caso del ejemplo resultará en una reducción considerable de la redundancia y una mejora importante en el grado de localidad. Esto es debido a que vamos a eliminar la redundancia sobre la variable x ya que se evaluarán los píxeles de dos filas en la función productora, previamente a la evaluación de la correspondiente fila de la función consumidora, almacenándose en la memoria volátil de manera temporal hasta empezar la evaluación de la siguiente fila de la función consumidora. En este último aspecto hemos incrementado la localidad, ya que la distancia entre la evaluación de un píxel en la función productora y su uso en la evaluación de la función consumidora se ha reducido considerablemente.

```
for( y = 0; y < 4; y++ )
       producer_buffer[2][5];
       for( py = y; py < y+2; py++ )
```

```
                for( px = 0; px < 4; px++ )
                        producer_buffer[py-y][px] = log(px + py);
        for( x = 0; x < 4; x++ )
                consumer[y][x] = producer_buffer[0][x]
                        + producer_buffer[0][x+1]
                        + producer_buffer[1][x]
                        + producer_buffer[1][x+1];
```

De manera gráfica podemos observar en la figura 5.6 la redundancia que se produce debido a la localización del buffer que almacena los píxeles evaluados de la función productora. Este buffer se encuentra dentro del bucle que itera sobre la variable y, por lo que para cada iteración deberán evaluarse dos filas de la función productora.

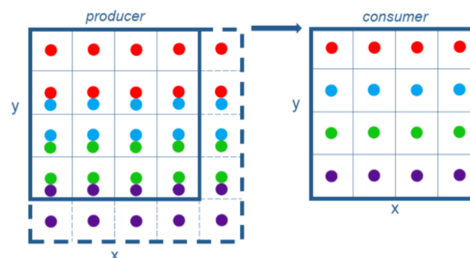


Figura 5.6: Esquema de evaluación de la función productora por filas

4.2.4 Evaluación por filas con búfer externo

Consiste en cambiar de lugar el vector `producer_buffer` que almacena los píxeles evaluados en la función productora para que en cada movimiento de filas en la evaluación de los píxeles de la función consumidora se reutilicen los píxeles de la fila correspondiente. Por lo que con este método vamos a modificar el lugar en el que se guardan los datos.

```
producer.store_root();
producer.compute_at(consumer, y);
consumer.realize(4, 4);
```

Por lo que tras haber mejorado la localidad de los datos en el apartado anterior, modificando el lugar de almacenamiento de los datos en el esquema de codificación conseguiremos evitar la redundancia, realizando el menor número posibles de cálculos en el procesador y haciendo un uso muy eficiente de la memoria cache. Obsérvese el cambio en la declaración del buffer que almacena la evaluación de los píxeles de la función productora:

```
producer_buffer[2][5];
for( y = 0; y < 4; y++ )
        for( py = y; py < y+2; py++ )
                if ( y > 0 && py == y )
                        continue;
                for( px = 0; px < 4; px++ )
                        producer_buffer[py&1][px] = log(px + py);
        for( x = 0; x < 4; x++ )
                consumer[y][x] = producer_buffer[y&1][x]
                        + producer_buffer[y&1][x+1]
                        + producer_buffer[(y+1)&1][x]
                + producer_buffer[(y+1)&1][x+1];
```

Se aprecia en la figura 5.7 que para cada iteración sobre las líneas de la función consumidora se evaluará únicamente una nueva línea de la función productora, y se aprovechará la línea evaluada en la iteración anterior:

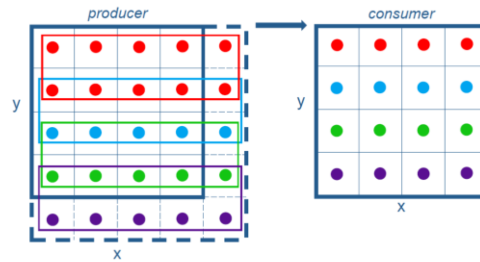


Figura 5.7: Esquema de evaluación de la función productora por filas con búfer externo

4.2.5 Evaluación por columnas con buffer externo

Aprovechando el efecto del buffer externo evaluaremos esta vez los píxeles necesarios de la función productora en cada iteración sobre la variable x de la función consumidora.

```
producer.store_root();
producer.compute_at(consumer, x);
consumer.realize(4, 4);
```

El código C equivalente sería el siguiente:

```
producer_buffer[2][5];
for( y = 0; y < 4; y++ )
    for( x = 0; x < 4; x++ )
        if ( y==0 && x==0 )
            producer_storage[y&1][x] = log(x + y);
        if ( y==0 )
            producer_storage[y&1][x+1] = log((x+1) + y);
        if ( x==0 )
            producer_storage[(y+1)&1][x] = log(x + (y+1));
        producer_storage[(y+1)&1][x+1] = log((x+1) + (y+1));

        consumer[y][x] = producer_buffer[0][x]
            + producer_buffer[0][x+1]
            + producer_buffer[1][x]
        + producer_buffer[1][x+1];
```

Se consigue reducir el número de accesos a memoria volátil ya que de cada cuatro píxeles de la función productora, un píxel es el que se acaba de computar (figura 5.8), por lo que es muy probable que este todavía se encuentre en el registro del procesador.

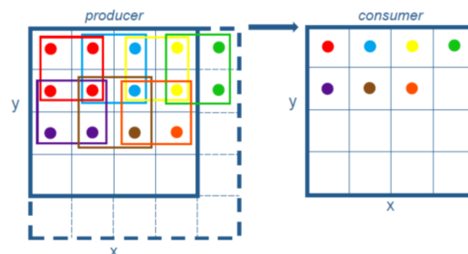


Figura 5.8: Esquema de evaluación de la función productora por columnas con búfer externo

4.2.6 Rendimiento de cada esquema

Se han analizado las estrategias básicas en la codificación de algoritmos multi-etapa. Modificando el lugar en el que se computan las sub-funciones y el lugar donde se almacenan estos valores conseguimos una mejora en la localidad de los datos y una reducción de la redundancia haciendo un uso eficiente de la memoria caché del procesador. No siempre será este el rendimiento óptimo, ya que si por ejemplo se dispone de una memoria caché reducida y un lento acceso a la memoria primaria quizás sea más óptimo hacer un mayor uso del

procesador (aumentar la redundancia) y realizar menos accesos a la memoria volátil. Se comparan los diferentes esquemas de evaluación de sub-funciones en la tabla 5.1.

	Evaluación con función insertada	**Evaluación al inicio**	**Evaluación por filas**	**Evaluación por filas con buffer externo**	**Evaluación por columnas con buffer externo**
Tamaño de la memoria temporal	0 px	25 px	10 px	10 px	10 px
Lecturas en memoria	0	64	64	64	48
Escrituras en memoria	16	41	56	41	41
Llamadas a la función log	64	25	40	25	25

Tabla 5 1: Rendimiento de los diferentes esquemas de procesado

5. MODOS DE COMPILACIÓN

Como se comentó con anterioridad, el código generado en el proceso de compilación de *HALIDE* puede volcarse directamente a memoria volátil o guardarse en la memoria de almacenamiento en forma de fichero objeto (*object file*). Lo más probable es que nuestra intención sea crear funciones utilizando *HALIDE* y llamar a estas desde nuestro programa, el cual realizará una tarea determinada. Una buena opción sería preparar un conjunto de funciones útiles utilizando *HALIDE* compilándolas de manera estática es sus respectivos ficheros objeto agrupando estos en una sola librería compartida (por ejemplo una librería *.so*).

5.1.1 Compilación en tiempo de ejecución

Por definición su ejecución será más lenta pero puede ser muy útil en el proceso de desarrollo o en ciertos casos específicos. Para funciones en las que exista un productor con un dominio definido (imagen en un buffer estático, es decir una función del tipo `Halide::Image`) se compilará en tiempo de ejecución y ejecutará el código generado de la siguiente manera:

```
Halide::Image image_result;
result.compile_jit();
result.realize(image_result);
```

Si la función a compilar no tiene un dominio acotado se deberá acotar este de la siguiente manera:

```
Halide::Image image_result;
result.compile_jit();
image_result = result.realize(4, 4);
```

5.1.2 Compilación estática

Se generará un fichero objeto con la función compilada en el formato adecuado para ser llamada desde otros programas desarrollados en lenguaje C (*C ABI-compatible*). Compilaremos de la manear habitual enlazando con la librería de *HALIDE* como haríamos para el caso anterior. Una vez finalizado el proceso de compilación ejecutamos el objeto generado y obtendremos el fichero objeto resultado de la compilación de *HALIDE*. Este fichero objeto es el que vincularemos con el programa que haga uso de la función optimizada.

```
std::vector<Halide::Argument> args(2);
args[0] = input;
args[1] = parameter_lambda;
result.compile_to_file("my_function", args);
```

Capítulo 6. Sistemas microprocesadores embebidos basados en distribuciones Android/Linux

Como se comentó en la introducción a la memoria, los sistemas embebidos sobre los que se va a trabajar utilizan el sistema operativo Android/Linux. Para poder desarrollar código de manera eficiente se hace necesario conocer las características de los sistemas para los que se está programando. En este capítulo se aportará un particular punto de vista sobre las distribuciones Android que nos ayudará al desarrollo de código nativo en estas plataformas.

1. Introducción

Actualmente existen multitud de sistemas microprocesadores embebidos que utilizan distribuciones Android como sistema operativo. Fácilmente cualquier individuo sin conocimientos técnicos relacionará Android con los teléfonos inteligentes. Pero no solo en este tipo de sistemas embebidos se implementan distribuciones Android/Linux, sino que también se pueden encontrar en diferentes tipos de sistemas como por ejemplo:

- **Entretenimiento**. Encontramos, entre otros, la gama de productos de NVIDIA SHIELD. Se trata de una plataforma de sistemas microprocesadores basados en un núcleo NVIDIA Tegra orientados al *gamming* y contenidos multimedia.
- **Contenidos multimedia.** Existen multitud de televisores que incorporan distribuciones Android.
- **Automoción.** Hoy en día el nivel de implicación de los sistemas microprocesadores en los automóviles es indiscutible. Estos sistemas están evolucionando y ya no solo se encargan del control y manejo de los parámetros del motor de combustión. Existen plataformas *software* como COQOS o arquitecturas *software* como AUTOSAR que contemplan la utilización de distribuciones Android.
- **Ingeniería biomédica**. Existen propuestas de utilización de Android conjuntamente con sistemas operativos en tiempo real (RTOS) para la implementación de instrumental médico.

El desarrollo de aplicaciones críticas para estos entornos implicará la optimización de código nativo por lo que resulta fundamental comprender la arquitectura de las distribuciones Android/Linux.

2. Android y Linux

Es común encontrar la definición de Android como un sistema operativo basado en un núcleo (*kernel*) de Linux. Sin embargo, existe cierta controversia en considerar si es un sistema

operativo basado en un núcleo Linux o si por el contrario es realmente un sistema operativo Linux.

El núcleo Linux que se implementa en las distribuciones Android es un tipo de **núcleo monolítico**. Por núcleo monolítico se entiende que es un núcleo formado por los procesos que ofrecen la funcionalidad propia del núcleo (acceso al hardware, planificador de procesos, comunicación básica entre procesos, manejo de memoria, etc.) y un conjunto de servicios que ofrecen la funcionalidad básica de un sistema operativo (controladores de los dispositivos, sistema de ficheros, gestión de llamadas al sistema, etc.). Por lo tanto existe una correspondencia entre el núcleo del sistema y el propio sistema operativo, ya que en este tipo de arquitecturas la totalidad del sistema operativo trabaja en el núcleo ejecutándose en modo protegido (*kernel mode*).

En el lado opuesto encontramos los **micronúcleos**, en los que solamente se ejecutan en modo protegido los componentes que ofrecen la funcionalidad básica en la planificación de procesos, manejo de interrupciones, acceso al hardware, etc.

Es decir, si para distribuciones GNU/Linux aceptamos la premisa de que el sistema operativo es el propio núcleo de Linux sobre el cual se incorporan una serie de aplicaciones del proyecto GNU, deberíamos aceptar también la definición de una distribución Android/Linux como un sistema operativo Linux sobre el que se incorporará un entorno de usuario basado en el proyecto Android. No sucedería lo mismo en distribuciones GNU/Hurd, ya que el núcleo GNU Hurd es de tipo micronúcleo con una arquitectura cliente-servidor, donde el sistema operativo se ejecuta también en el espacio de usuario (fuera del núcleo).

3. MODIFICACIONES EN EL NÚCLEO VANILLA

Los núcleos en su forma “*vanilla*” son los que se pueden obtener a través de [11]. Es decir son los núcleos originales sobre los que cada distribución realiza sus modificaciones para adaptarlo a sus necesidades. Los cambios que encontramos en el núcleo Linux de una distribución Android son los que habitualmente se realizan en cualquier modificación del *kernel* para adaptarlo a sistemas embebidos (aproximadamente unos 250 parches). Los cambios se encuentran en el manejo de la memoria, el sistema de trazado (*logger*), manejo de la energía, incremento de la seguridad de red (*paranoid*), sistema de intercomunicación de procesos (*Binder*), etc.

3.1 TÉRMINOS DE LICENCIA

Los núcleos Linux se distribuyen con una licencia GNU General Public License (GPL) versión 2. Cuando se distribuya software basado en código fuente con licencia GPL se está obligado a hacer público el código fuente modificado. Muchas modificaciones que encontramos en distribuciones Android se originan en *evitar* las licencias GPL. Como muchos autores afirman [16], las intenciones de Google han sido aislar el software sujeto a licencias GPL en el núcleo del sistema (por lo tanto solo se tiene la obligación de liberar el código del *kernel*), y aplicar otro tipo de licencias más favorables a los intereses de la compañía en el espacio de usuario. Se encuentran ejemplos como la reimplementación de la biblioteca estándar de C (*libc*) que actúa como interface entre el núcleo y el espacio de usuario, o la reimplementación de una máquina virtual (*Dalvik*) en el que aseguran un diseño en “*sala limpia*”, es decir, un método de copia basado en la ingeniería inversa y la reescritura de código sin infringir términos de copyright. Otro caso interesante es el de las *capas de abstracción de hardware* (*HAL*). Al tratarse de un núcleo monolítico los controladores de dispositivo se implementarán en el núcleo del sistema (licencia GPL) pero a través de la implementación de una funcionalidad básica. El acceso funcional a los controladores se realizará a través de librerías compartidas que se encuentran en el espacio de usuario (*user space*), las cuales se distribuirán con la licencia que el fabricante determine.

4. ENTORNO DE USUARIO

El entorno de usuario se basa en la máquina virtual que se encargará de interpretar las aplicaciones desarrolladas en Java (con el formato propio de Dalvik). En este entorno disponemos de APIs para acceder a los recursos hardware y para interactuar con el resto del entorno de usuario. A modo ilustrativo, y sin entrar en detalle, se muestra la arquitectura para el acceso a un dispositivo hardware (figura 6.1). En color naranja se encuentra la aplicación que ofrecerá a otras aplicaciones el servicio de acceso a la cámara del dispositivo a través de la definición de la API. Por debajo, en color morado, encontramos la capa de abstracción de hardware que será la que se deba desarrollar específicamente para el modelo de dispositivo. En sombreado verde se encuentra la interfaz nativa de Java que actuará de enlace, a través del uso de espacios de memoria compartida, con el resto del sistema.

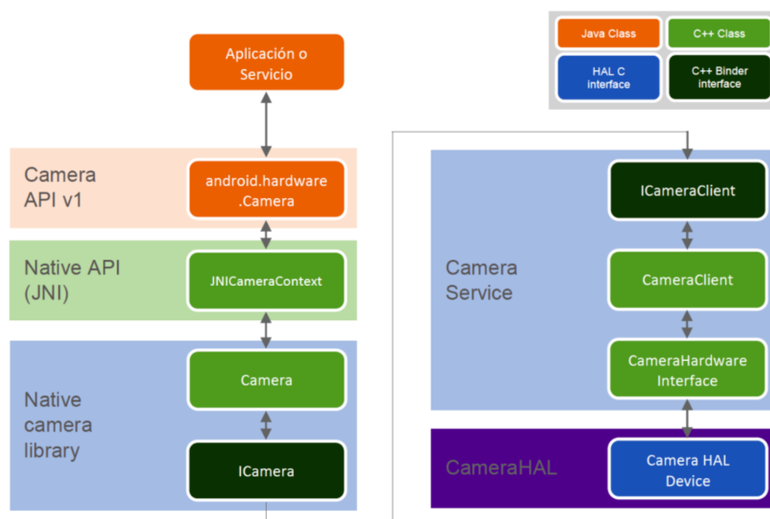


Figura 6.1: Arquitectura de acceso al hardware en el entorno Android

5. CONCLUSIONES RELATIVAS AL PROCESO DE OPTIMIZACIÓN

Conocer la arquitectura de una distribución Android/Linux nos permitirá, primero que nada, saber realmente para que plataforma estamos desarrollando, y en segundo lugar, realizar optimizaciones de código de una manera más efectiva. Saber que Android es un sistema operativo Linux nos permitirá situarnos en el espacio de usuario cuando sea necesario, y poder utilizar herramientas como la consola de comandos (shell), acceso remoto a consola (ssh), monitorización de procesos (top), compiladores (gcc), etc. Algunos dirán que no es necesario hacer uso de todas estas aplicaciones y que se puede trabajar con código nativo ejecutándolo desde la instalación de la aplicación del entorno de Android (*.apk*). Esto será cierto para los casos en que *todo vaya bien*.

En el desarrollo de código nativo para distribuciones Android hay que realizar diversas pruebas para evaluar el rendimiento de la implementación. Si estas pruebas se realizan de la manera *tradicional* habrá que desarrollar las interfaces de código haciendo uso de JNI y desarrollar la implementación JAVA correspondiente. Muchas veces la optimización conllevará la modificación de la estructura de código y, por tanto, modificaciones en la aplicación desarrollada en JAVA. Desde mi punto de vista, este no es el proceso más eficiente en el desarrollo de código nativo para distribuciones Android ya que se invierte demasiado tiempo en códigos accesorios que no son objeto de la optimización. El proceso que se recomienda es trabajar en el espacio de usuario de Linux y realizar, mediante las aplicaciones necesarias (*ssh*, *shell*, *top*, *vi*, *gcc*, etc.), toda la batería de pruebas que se necesite, trabajando de este modo de una manera más rápida y eficiente. Este es el método utilizado en este proyecto para la evaluación de la implementación realizada para distribuciones Android/Linux.

Capítulo 7. Desarrollo del modelo de referencia del algoritmo

En este capítulo se realizará, utilizando Matlab, la simulación de un algoritmo de eliminación de ruido basado en el concepto de variación total utilizando las técnicas de programación habituales. Se detallarán una serie de simplificaciones sobre al algoritmo presentado con la finalidad de hacer más sencillas las implementaciones. La simulación realizada se utilizará como modelo de referencia para validar el correcto funcionamiento de las futuras implementaciones.

1. DESCRIPCIÓN Y SIMPLIFICACIÓN DEL ALGORITMO

Tal como se indicó anteriormente, se va a implementar un filtro para la eliminación de ruido Gaussiano utilizando el algoritmo ROF basado en la variación total (TV). Este algoritmo se ha explicado en apartados anteriores, llegando a la siguiente ecuación:

$$u_{i,j}^{n+1} = u_{i,j}^{n} + dt\left[\nabla_x^-\left(\frac{\nabla_x^+ u_{i,j}^n}{\sqrt{\left(\nabla_x^+ u_{i,j}^n\right)^2 + \left(m\left(\nabla_y^+ u_{i,j}^n, \nabla_y^- u_{i,j}^n\right)\right)^2}}\right) + \nabla_y^-\left(\frac{\nabla_y^+ u_{i,j}^n}{\sqrt{\left(\nabla_y^+ u_{i,j}^n\right)^2 + \left(m\left(\nabla_x^+ u_{i,j}^n, \nabla_x^- u_{i,j}^n\right)\right)^2}}\right)\right] + dt\lambda\left(r_{i,j} - u_{i,j}^n\right)$$

Esta expresión corresponde a la discretización del método de gradiente descendente para la obtención de la solución a la ecuación de Euler-Lagrange que resuelve el problema de minimización que se planteó en el apartado correspondiente. Se realizará un número determinado de iteraciones aplicando la expresión anterior a la imagen que se desea filtrar. Para t se adoptará un valor pequeño ($t = 0.5$). El valor de λ debería ser calculado para cada iteración, no obstante para simplificar la implementación utilizaremos un valor fijo en todo el proceso. Así mismo, para realizar el número de iteraciones estrictamente necesario debería calcularse la diferencia en términos de energía entre la solución de la iteración n y la iteración $n - 1$, fijándose así el valor de paro del método de gradiente descendiente. En nuestro caso se aplicará un número fijo de iteraciones para simplificar la implementación. Por otra parte, $m(a, b) = \left(\frac{\operatorname{sgn} b + \operatorname{sgn} a}{2}\right)\min(|a|, |b|)$ se substituirá por la derivada parcial por la derecha correspondiente, también por motivos de simplificación. Se ha evaluado el impacto de todas estas simplificaciones y para fines de desarrollo pueden asumirse los errores cometidos, obteniéndose unos resultados muy aproximados al resultado ideal.

2. SIMULACIÓN Y RESULTADOS

A partir de las consideraciones anteriores se realiza la implementación del algoritmo en Matlab con el código del Anexo 1. Se realizan los cálculos por etapas, guardando estas una relación

con la posterior implementación en lenguaje C. De esta manera será más fácil depurar la implementación realizada y en caso de errores se podrán comparar los resultados obtenidos en cada etapa de la implementación con los obtenidos en la simulación. Es decir, mediante la simulación realizada en Matlab se ha asegurado el correcto funcionamiento del algoritmo mediante la validación de los resultados obtenidos. Esta simulación será el modelo de referencia mediante el cual validaremos las implementaciones realizadas posteriormente utilizando la tecnología determinada (en nuestro caso serán lenguaje C++ y lenguaje HALIDE) a través de la comparación de los resultados.

Se realizan pruebas con diferentes imágenes contaminadas por ruido Gaussiano, obteniéndose a la salida del algoritmo una imagen filtrada en la que se mantienen los bordes de los conjuntos de nivel (regiones en la imagen con unos mismos valores). Se muestra a continuación una imagen contaminada por ruido Gaussiano (figura 7.1) y el resultado obtenido tras aplicar el algoritmo implementado (figura 7.2).

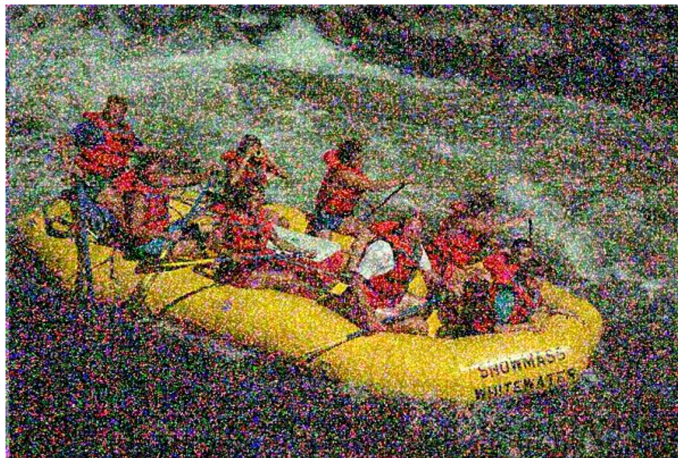


Figura 7.1: Imagen contaminada por ruido Gaussiano

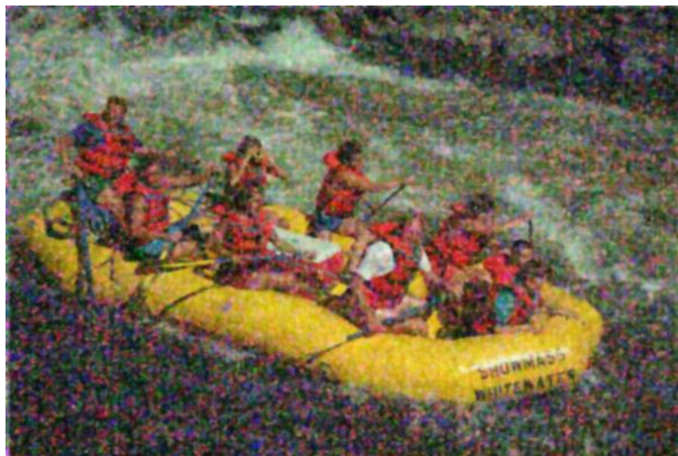


Figura 7.2: Imagen obtenida tras aplicar el algoritmo

Capítulo 8. Implementación del algoritmo mediante el uso de las técnicas habituales

En este capítulo se implementará un algoritmo de eliminación de ruido basado en el concepto de variación total utilizando las técnicas de programación habituales. Se realizarán tres implementaciones utilizando librerías específicas optimizadas (OpenCV, Eigen y Armadillo). Se analizarán las opciones de uso de técnicas avanzadas de programación. Finalmente se compilará cada implementación para diferentes plataformas de microprocesadores y se analizarán sus respectivos rendimientos.

1. OBJETIVO

En esta etapa del proyecto se implementará el algoritmo modelado en la simulación haciendo uso de las técnicas de programación más habituales. Como se ha comentado en otros apartados, para el desarrollo de una implementación optimizada podemos hacer uso de:

- **Librerías** optimizadas de terceros.
- Optimización manual del código fuente mediante **técnicas de programación** (es decir, definir el esquema óptimo del programa y codificarlo haciendo uso de vectorización con instrucciones *SIMD*, paralelización de bucles, correcto uso de *bufferes* temporales, etc.)

En este apartado se desarrollará una implementación optimizada haciendo uso de librerías de terceros. No se realiza una optimización manual del programa debido a los costes de tiempo que ello supondría, no obstante se expondrán en el apartado correspondiente situaciones en nuestra implementación en las que resultaría interesante realizar una optimización manual de manera local en determinadas funciones.

Para cada implementación realizada se validará el correcto funcionamiento del programa comparando el resultado con el modelo de referencia desarrollado en Matlab. Se realizará un estudio comparativo de cada una de las implementaciones mostrando el tiempo de ejecución para diferentes tamaños de imagen, obteniendo de este modo el rendimiento de cada implementación en función de las dimensiones de la imagen de entrada. Este estudio se realizará para cada uno de los diferentes sistemas microprocesadores definidos para poder conocer el funcionamiento de cada implementación de acuerdo al sistema microprocesador en el que se ejecuta.

2. USO DE LIBRERÍAS

Se realizarán tres implementaciones en lenguaje C++ haciendo uso de las librerías más utilizadas en procesamiento de imagen. En un primer momento se desea realizar los desarrollos de manera íntegra con cada librería respectiva, no obstante se comprobará que estas no serán las implementaciones más eficientes y que habrá que hacer uso de *OpenCV* en todas ellas, de manera que tendremos los siguientes esquemas de librerías:

- **OpenCV**. Se realiza la implementación haciendo uso únicamente de la librería de visión artificial *OpenCV*.
- **OpenCV + Eigen**. Se optimiza la implementación anterior realizando ciertos cálculos matriciales utilizando la librería de álgebra lineal *Eigen*.
- **OpenCV + Armadillo + OpenBLAS**. Se optimiza la implementación realizada con *OpenCV* realizando ciertos cálculos matriciales utilizando la librería de álgebra lineal Armadillo (con *OpenBLAS*).

Se definirá la estructura del proyecto y se describirá brevemente la codificación de cada una de las implementaciones. Se ejecutaran para diferentes tamaño de imagen y se definirá el tiempo de ejecución de cada una de las implementaciones en función de la dimensión de la imagen de entrada. Se valorarán individualmente otros aspectos como el coste de desarrollo utilizando cada librería, la facilidad de interacción con OpenCV, los procesos de compilación para cada una de ellas, etc.

2.1 ESTRUCTURA DE FICHEROS Y PROCESO DE COMPILACIÓN

Para trabajar de manera adecuada se diseña un proyecto tipo sobre el que se trabajará. Se implementan los filtros utilizando las respectivas librerías en ficheros de texto separados con la extension "*.cpp"* dentro de la carpeta *src*, además de los respectivos ficheros de cabeceras con la extensión "*.h*" en la carpeta *include/denoise*. Este código fuente se compilará en los respectivos *ficheros objeto*, los cuales se juntarán para crear una única librería que contendrá las tres implementaciones ("*libdenoise*"). Esta librería se creará en la carpeta *lib* con la extensión "*.so*" (librería compartida). Se codificará un programa que ejecute las pruebas de tiempo llamando a las respectivas funciones implementadas, por lo que cuando se compile este ejecutable se enlazará con la libería generada (con el *flag* "*-ldenoise*" e indicando su ruta a la carpeta *lib*).

Para automatizar todo el proceso se utilizará la herramienta *GNU Make* [6]. Esta aplicación permite automatizar los procesos de compilación definiendo este proceso en un fichero "*Makefile"*. Previo al inicio del proceso de compilación debemos disponer de todas las librerías necesarias compiladas en la máquina. Se recomienda la descarga del código fuente de cada una de estas librerías y su compilación en la máquina, en particular *OpenBLAS*. Para facilitar la modificación de las rutas a las localizaciones de las librerías se ha creado un fichero "*Makefile.inc"* que contiene las variables con los respectivos directorios, los cuales se tendrán que modificar dependiendo de donde se encuentren en la máquina en la que se desea compilar. Este es el único fichero que hay que modificar para compilar y ejecutar el estudio comparativo de las librerías.

Finalmente se ha creado un script en *BASH* para la llamada al ejecutable generado. Este script se encarga de leer el formato de la llamada y de añadir al entorno de ejecución la ruta a la librería generada antes de llamar al ejecutable, ya que se ha vinculado con la librería compartida (las librerías compartidas deben ser accesibles, en nuestro caso mediante la variable de entorno *LD_LIBRARY_PATH*).

Tras el proceso de compilación tendremos la siguiente estructura de ficheros:

`/include` – *Carpeta con las ficheros de cabecera.*
```
        benchmark.h
        /denoise
```

```
            denoise_arma.h
            denoise_eigen.h
            denoise_opencv.h
/lib
        libdenoise.so – Librería compartida con las implementaciones
/src – Carpeta en la que se encuentra el código fuente
        denoise_arma.cpp
        denoise_eigen.cpp
        denoise_opencv.cpp
        test_denoise.cpp
Makefile
Makefile.inc – Contiene las variables de compilación
test – Fichero ejecutable que inicia el programa
tes_denoise
```

Para ejecutar "*test*" tenemos que seguir el siguiente formato de llamada:

```
./test [PATH_TO_IMAGE] [N]
```

Donde `N` es la variable que selecciona la función a evaluar. Para ejecutar el filtro implementado con *OpenCV* llamaremos con `N=1`, para el filtro implementado con *Eigen* llamaremos con `N=2` y para el filtro implementado con *Armadillo* con `N=3`.

2.2 IMPLEMENTACIÓN CON LIBRERÍA OPENCV

Se realiza la implementación correspondiente al fichero fuente "denoise_opencv.cpp" disponible en el anexo 2.

2.2.1 Codificación

El algoritmo se basa en el cálculo de gradientes y operaciones básicas entre matrices elemento a elemento. Para el cálculo de los gradientes se definirá un filtro de coeficientes $[-1\,1]$ y $[1-1]$ para las derivadas discretas por la derecha y por la izquierda, respectivamente, en el eje x. Para el eje y se utilizarán como coeficientes la trasposición de las matrices utilizadas en el eje de abscisas. Estos filtros se convolucionarán con la imagen haciendo uso de la función *filter2D* disponible en la librería *OpenCV*:

```
filter2D(input_n, fx, ddepth, kernelx, anchorx_r, delta, BORDER_REPLICATE);
filter2D(input_n, fy, ddepth, kernely, anchory_r, delta, BORDER_REPLICATE);
```

Se indican como parámetros la matriz de entrada y la de salida, así como el formato numérico de la matriz de salida, los coeficientes de los filtros y la posición relativa de ellos. Esta posición relativa nos indica qué coeficiente corresponde a la posición 0, por lo que podemos hacer uso de un único filtro de coeficientes $[-1\,1]$ y situar la posición cero en el coeficiente -1 para el gradiente por la derecha o en el coeficiente 1 para el gradiente por la izquierda. Los últimos dos términos nos indican si deseamos añadir una componente continua al resultado y el comportamiento en los bordes.

2.2.2 Evaluación en GNU/Linux con arquitectura x86-64

Se ejecuta la implementación realizada para imágenes de diferentes medidas, con la intención de caracterizar el coste en términos de tiempo de ejecución en función de las dimensiones de la imagen de entrada. Se obtiene el resultado mostrado en la figura 8.1.

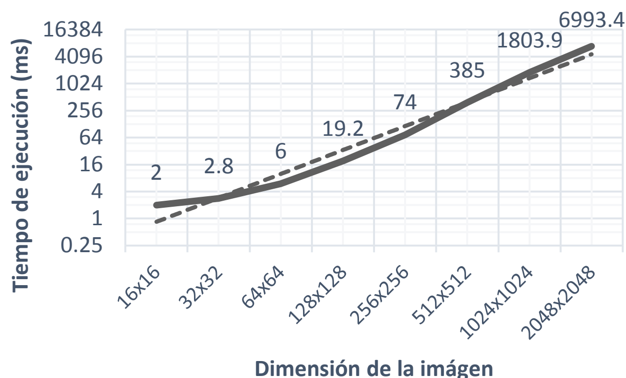


Figura 8.1: Rendimiento utilizando OpenCV en función de las dimensiones para arquitecturas x86-64

Se observa que el tiempo de ejecución es linealmente dependiente con las dimensiones de la imagen, aproximadamente. Si bien es cierto que para ciertas dimensiones la librería realiza los cálculos con un rendimiento mayor.

2.2.3 Evaluación en Android/Linux con arquitectura ARMv7

Evaluamos el comportamiento de la implementación realizada ejecutándose en los sistemas microprocesadores embebidos indicados en la introducción de la memoria. Para el tiempo de ejecución en función de las dimensiones de la imagen se obtiene el mismo comportamiento que para sistemas *GNU/Linux* sobre procesadores *x86-64*, es decir, una respuesta del tiempo de ejecución linealmente dependiente con el número de píxeles de la imagen de entrada.

Se comparan los diferentes resultados para los tres sistemas evaluados en la figura 8.2, los cuales son los esperados de acuerdo a las características técnicas de cada sistema embebido. Se limitan las dimensiones de la imagen debido al exceso de tiempo consumido para procesarlas.

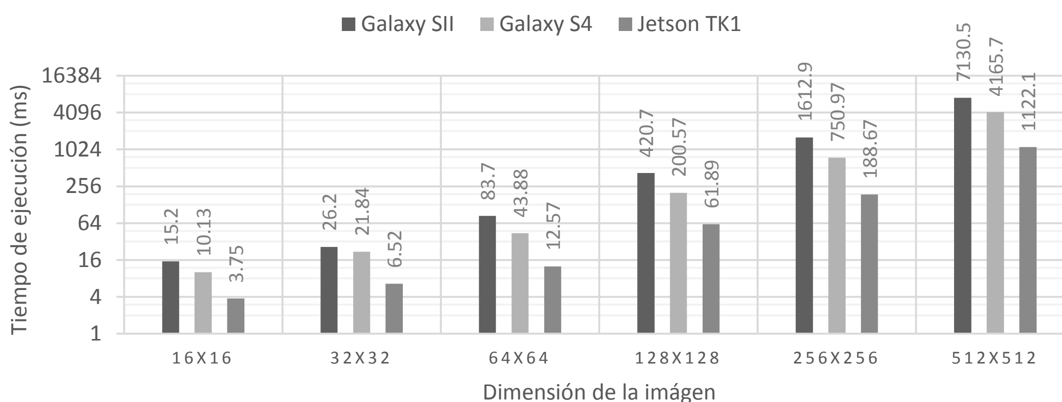


Figura 8.2: Rendimiento utilizando OpenCV en función de las dimensiones para arquitecturas ARM

2.2.4 Valoraciones

La librería OpenCV es la librería por excelencia en el campo de la visión artificial. Dispone de una API muy bien documentada y se encuentra multitud de información en la red. En muchos casos en los que tengamos dudas acerca de cómo abordar algún problema podremos acudir a multitud de foros en los que seguramente encontraremos problemas similares a los nuestros. Dispone además de multitud de funciones enfocadas a la visión artificial, por lo que su uso resulta muy fácil y las implementaciones se hacen rápidamente. No obstante, según se deprende de los resultados, el tiempo de ejecución de las aplicaciones desarrolladas con esta librería no es el ideal.

Por lo que respecta a la compilación cruzada para distribuciones Android, no se han encontrado dificultades importantes. Se debe compilar la librería *OpenCV* para Android/Linux sobre arquitecturas ARMv7. Este proceso no resulta complicado ya que se facilita un fichero *toolchain* el cual, utilizando *CMake*, nos permitirá seleccionar todas las características que deseamos en el proceso de compilación (versión de Android, utilización de *SIMD Neon*, *TBB*, etc.).

2.3 IMPLEMENTACIÓN CON LIBRERÍAS OPENCV Y EIGEN

Se realiza la implementación correspondiente al fichero fuente "denoise_eigen.cpp" disponible en el anexo 3.

2.3.1 Codificación

Se realiza la implementación utilizando únicamente la librería *Eigen*. Se observa que el resultado es mucho más lento que utilizando OpenCV, contradiciendo a los análisis expuestos en el capítulo dedicado a las librerías. Se localizan las zonas de mayor consumo de tiempo y que corresponden al cálculo de la convolución.

No se han encontrado métodos eficientes para realizar la convolución necesaria en el algoritmo, es decir, Eigen no dispone de una función optimizada para realizar la convolución de dos imágenes. Se realiza la siguiente implementación de la convolución utilizando *Eigen*:

```
for ( row = KSizeX; row < SizeX-KSizeX; row++ ) {
    for ( col = KSizeY; col < SizeY-KSizeY; col++ ) {
        Scalar accumulation = 0;
        Scalar weightsum = 0;
        for ( i = 0; i < KSizeX; i++ ) {
           for ( j = 0; j < KSizeY; j++ ) {
               Scalar k = I(row+i, col+j);
               accumulation += k * kernel(i,j);
               weightsum += kernel(i,j);
           }
        }
        O(row,col) = Scalar (accumulation/weightsum);
    }
}
return O
```

A simple vista se observa que nunca esta implementación puede ser óptima, por lo que habría que buscar la manera de *eludir* los bucles. Si el lector tiene interés puede acudir a [10], donde encontrará alternativas para una implementación más eficiente de esta función.

No obstante, y por motivos de tiempo, se decide hacer uso de la función de *OpenCV* para realizar el filtrado debido a su excelente optimización. Esto nos obliga a trabajar con las dos librerías al mismo tiempo, es decir, nos obliga a tener que *jugar* con representaciones diferentes de la misma imagen. Sin embargo hacer esto con *OpenCV* y *Eigen* resulta sencillo y no supone un gran problema. Declararemos e inicializaremos las matrices de transición en *OpenCV* y seguidamente haremos un *mapeado* del contenido de estas matrices utilizando la clase *Map* de *Eigen*. Esta clase está pensada para trabajar con búferes externos de memoria, por lo que mediante el puntero a los píxeles del objeto *Mat* de *OpenCV* conseguimos trabajar sobre el mismo conjunto de datos utilizando las dos librerías.

Es decir, para operar utilizando las dos librerías sobre una misma imagen partiremos del puntero a los datos del objeto *Mat* y crearemos el nuevo objeto *Map* cuyos datos apuntarán a la misma imagen. Se ha resaltado en color el paso del puntero.

```
Mat Module(src.rows,src.cols,CV_32F);
Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic,
      Eigen::RowMajor> > Module_mappedMat ((float *)Module.data, src.rows,
      src.cols);
```

Mediante este método realizaremos la totalidad de las operaciones utilizando el entorno de *Eigen* a excepción de la convolución, la cual la realizaremos sobre el entorno *OpenCV*.

2.3.2 Evaluación en GNU/Linux con arquitectura x86-64

Se realiza el mismo banco de pruebas para la nueva implementación evaluando la respuesta del filtro en términos de tiempo de ejecución en función de las dimensiones de la imagen.

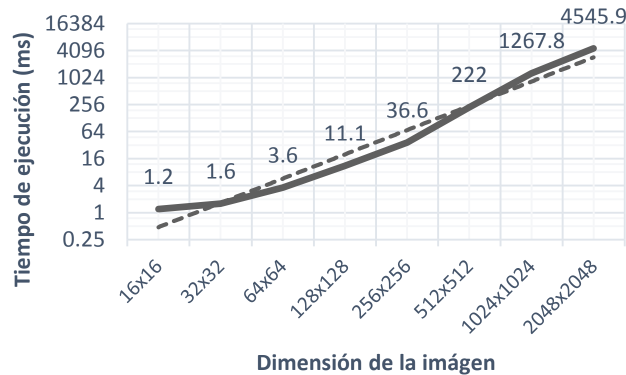


Figura 8.3: Rendimiento utilizando OpenCV+Eigen en función de las dimensiones para arquitecturas x86-64

En la figura 8.3 se observa que de nuevo el tiempo de ejecución es linealmente dependiente con las dimensiones de la imagen, aproximadamente. En este caso también se observa que para ciertas dimensiones la librería realiza los cálculos con un rendimiento mayor.

2.3.3 Evaluación en Android/Linux con arquitectura ARMv7

Se evalúa la implementación con *Eigen* en los tres sistemas microprocesadores embebidos que se han indicado. Nuevamente se obtiene el mismo comportamiento que para sistemas *GNU/Linux* sobre procesadores *x86-64*, es decir, una respuesta del tiempo de ejecución linealmente dependiente con el número de píxeles de la imagen de entrada.

Se comparan los diferentes resultados para los tres sistemas evaluados en la figura 8.4, los cuales son los esperados de acuerdo a las características técnicas de cada sistema embebido. Se limitan las dimensiones de la imagen debido al exceso de tiempo consumido para procesarlas.

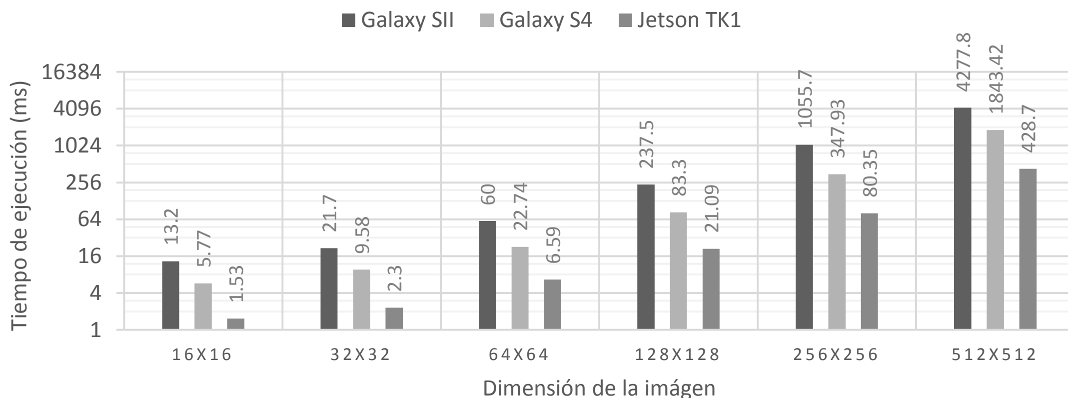


Figura 8.4: Rendimiento utilizando OpenCV+Eigen en función de las dimensiones para arquitecturas ARM

2.3.4 Valoraciones

Se observa una reducción considerable en el tiempo de ejecución, lo que concuerda con la idea que teníamos sobre el rendimiento de *Eigen* de acuerdo con las pruebas de velocidad

expuestas en el apartado correspondiente a las librerías. Sin embargo nos queda un *sabor agridulce* a pesar de los buenos resultados. La no disponibilidad de una función en la API de *Eigen* para realizar la convolución de una manera eficiente nos ha obligado a tener que acudir a otras herramientas. Si esta función es de uso recurrente por el programador sería posible buscar el método de implementarla de manera eficiente utilizando la librería *Eigen*, pero si no es el caso habría que valorar si el coste de desarrollo es asumible.

Por otro lado, la interoperabilidad de *Eigen* con otras librerías es un punto fuerte. *Eigen* no es una librería de visión artificial, por lo que muchas funciones que encontramos implementadas en OpenCV de manera eficiente no las encontraremos en la API de *Eigen*. Como hemos visto es posible trabajar con las dos librerías cómodamente por lo que desarrollar utilizando esta vía es una buena elección para generar código optimizado.

La compilación cruzada para diferentes arquitecturas y sistemas operativos resulta muy sencilla para el caso de la librería *Eigen*, ya que al tratarse de una librería de plantillas de C++ no se debe pre-compilar para ninguna arquitectura, es decir, únicamente se necesitan los archivos de cabecera de *Eigen*.

2.4 IMPLEMENTACIÓN CON LIBRERÍAS OPENCV Y ARMADILLO+OPENBLAS

En vistas a los prometedores resultados expuestos en el análisis de rendimiento realizado en el apartado dedicado a las librerías se realiza la implementación utilizando *Armadillo+OpenBLAS*. Esta implementación la encontramos en el fichero fuente “denoise_arma.cpp” disponible en el anexo 4.

2.4.1 Codificación

Al igual como sucedió en el caso anterior (implementación utilizando *Eigen*), no se ha encontrado en la API de *Armadillo* método alguno para la realización eficiente de la convolución. Por lo tanto se aplicará el mismo método que en el apartado anterior, es decir, combinar una libraría de cálculo que operará las matrices de una manera más optimizada con las funciones disponibles en OpenCV para la implementación de algoritmos de visión artificial.

El método de construcción de los objetos *fmat* de *Armadillo* a partir de un puntero a un búfer externo es muy similar al que utilizamos en *Eigen*:

```
Mat W(src.rows,src.cols,CV_32F);
arma::fmat arma_W( W.ptr<float>(), W.rows, W.cols, false, true );
```

Cabe poco que añadir ya que la sintaxis resulta muy sencilla. Hay que señalar que los últimos dos booleanos indican como tratar el búfer. El primero indica si deseamos copiar el contenido del búfer externo (*true*) o si por el contrario queremos leer directamente el contenido del búfer (*false*). El segundo booleano se utilizará para respetar el tamaño del búfer al que apunta el puntero y evitar que durante las operaciones se modifique el tamaño asignado en la declaración del búfer (*true*), o si por el contrario permitimos que pueda ser modificado (*false*).

2.4.2 Evaluación en GNU/Linux con arquitectura x86-64

Se realizan las pruebas de coste en términos de tiempo de ejecución en función de las dimensiones de la imagen de entrada. Se vuelve a obtener una respuesta que aproximadamente es lineal con el número de píxeles de la imagen de entrada. También se observa en la figura 8.5 que para ciertas dimensiones la librería realiza los cálculos con un rendimiento mayor.

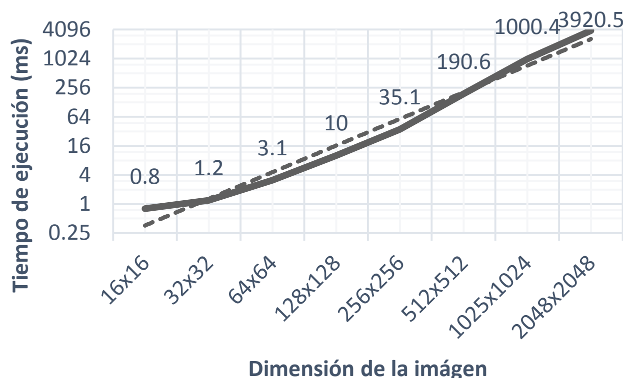


Figura 8.5: Rendimiento utilizando OpenCV+Armadillo en función de las dimensiones para arquitecturas x86-64

2.4.3 Evaluación en Android/Linux con arquitectura ARMv7

Se desea compilar la implementación realizada con la librería *Armadillo* para distribuciones Android/Linux sobre procesadores ARMv7, sin embargo no ha sido posible realizarse la compilación en un tiempo prudencial. Para poder compilar la implementación realizada para una arquitectura ARMv7 hay que disponer de las librerías previamente compiladas para la misma arquitectura. En el caso de *Armadillo+OpenBLAS* se debe realizar una compilación cruzada para sistemas embebidos Android/Linux con arquitectura ARMv7, en primera instancia del código fuente de *OpenBLAS* y seguidamente del código fuente de *Armadillo* vinculando con el resultado de la compilación cruzada de *OpenBLAS*. Este proceso no es arbitrario y conlleva una inversión de tiempo importante. En el caso de *OpenBLAS* se facilitan en el sitio web oficial paquetes pre-compilados para diferentes arquitecturas, entre estas: Linux ARMv6/ARMv7 y Android ARMv5tel. Sabiendo que existe la posibilidad de compilación para Linux sobre ARMv7 podríamos asegurar que se puede realizar una compilación para Android/Linux sobre ARMv7, sin embargo habría que estudiar bien los cambios que se deberían realizar de acuerdo a lo explicado en el apartado dedicado a las distribuciones Android. Por lo que respecta a *Armadillo* hay que tener un buen conocimiento acerca del funcionamiento de esta librería para poder realizar la compilación cruzada y así configurar los parámetros de la librería previamente a la compilación (por ejemplo la arquitectura ARMv7 es de 32 bits, por lo que se debe configurar la librería *Armadillo* para que no trabaje con tipos de 64 bits, entre otros parámetros).

2.4.4 Valoraciones

Tras la comparación de los resultados se validan nuevamente las conclusiones realizadas en el apartado dedicado a las librerías en el que se afirmaba que *Armadillo+OpenBLAS* era la opción más veloz del conjunto de librerías que se analizaron. En este caso se ha obtenido una implementación más optimizada que la obtenida con *Eigen*, si bien la mejoría de rendimiento no es muy acuciada ya que las cuatro convoluciones continúan operándose utilizando *OpenCV*. Por lo que respecta a su utilización es una librería muy práctica ya que dispone de métodos para trabajar con punteros a búferes de memoria externos y permite operar directamente sobre ellos. Esta interoperabilidad, al igual que sucedió con *Eigen*, hace que su uso sea muy recomendable conjuntamente con librerías específicas. Utilizar la librería *OpenCV* para métodos específicos de visión artificial y realizar las operaciones matriciales necesarias con *Armadillo* es una combinación perfecta para desarrollar implementaciones optimizadas. Sin embargo se han detectado dificultades en el proceso de compilación cruzada para distribuciones Android sobre arquitecturas ARMv7, lo cual puede limitarnos enormemente en el desarrollo de aplicaciones multiplataforma.

2.5 COMPARACIÓN DE RESULTADOS

Se normalizan los resultados de manera que se obtiene la gráfica comparativa mostrada en la figura 8.6 en la que se ilustra el incremento de rendimiento en términos de tiempo de ejecución. Se muestran a continuación la gráfica comparativa correspondiente a las pruebas realizadas para sistemas operativos GNU/Linux en arquitecturas x86-64.

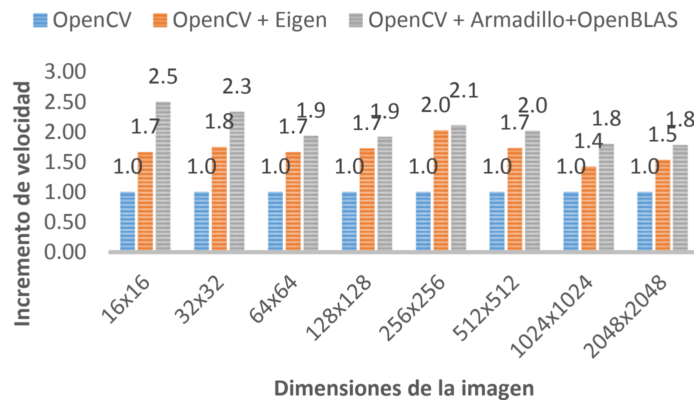


Figura 8.6: Incremento del rendimiento según la libreria utilizada en arquitecturas x86-64

Para la normalización se ha dividido el valor máximo obtenido en cada tamaño de imagen por el valor de las respectivas medidas de tiempo de las otras librerías. Los mejores resultados los obtenemos para la combinación de *OpenCV + Armadillo+OpenBLAS*, aunque para algunos tamaños de imagen este rendimiento es comparable al de la librería *Eigen*.

Para el caso de las implementaciones realizadas para sistemas operativos Android/Linux sobre arquitecturas ARMv7 se obtiene la gráfica comparativa mostrada en la figura 8.7, la cual es similar a la obtenida para sistemas operativos GNU/Linux sobre arquitecturas x86-64.

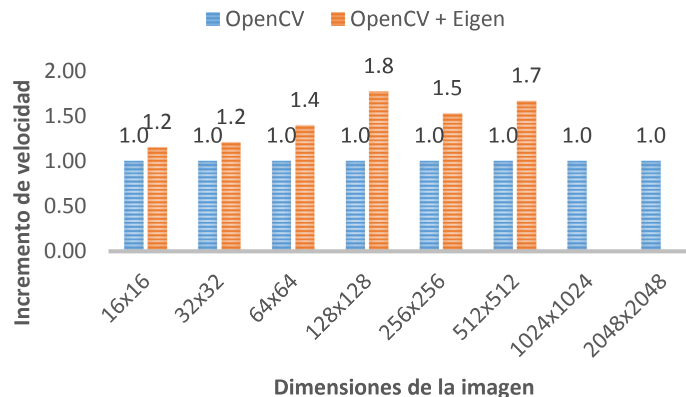


Figura 8.7: Incremento del rendimiento según la libreria utilizada en arquitecturas ARM (Galaxy SII)

3. USO DE OTROS MÉTODOS HABITUALES DE OPTIMIZACIÓN

Se desprende del análisis anterior, acerca de implementaciones optimizadas haciendo uso de librerías externas, que resulta difícil definir un modelo optimizado de programación para aplicaciones de visión artificial en general. En nuestro caso se ha requerido hacer uso de al menos dos librerías de terceros para obtener una implementación con un reducido impacto en el rendimiento del sistema, obteniendo con las liberías *Eigen* y *Armadillo* una optimización considerable en las operaciones básicas entre matrices.

Si esta implementación requiriese de una reducción aún mayor del tiempo de ejecución se debería optimizar la codificación de manera manual haciendo uso de técnicas de programación como vectorización (utilización de funciones intrínsecas, SIMD), paralelización (diseño del esquema de cálculo, *pool* de *threads*), mejora del empleo de memoria volátil (incremento en el uso de la memoria cache, programación parametrizada por bloques), etc. Programar todo el algoritmo utilizando estas técnicas puede significar un incremento en el tiempo de desarrollo inasumible, sin embargo se puede analizar los puntos críticos de nuestro código e implementar estas partes utilizando las técnicas comentadas. Para el caso del filtro de eliminación de ruido implementado encontramos que la única parte de nuestro código que no hemos podido mejorar es la correspondiente a las convoluciones. Por lo que se debería de analizar el código fuente de la función que hemos utilizado (*filter2D* de *OpenCV*) y deducir de manera crítica si vamos a ser capaces de batir esa implementación.

Capítulo 9. Implementación del algoritmo utilizando el lenguaje HALIDE

En este capítulo se implementará un algoritmo de eliminación de ruido basado en el concepto de variación total utilizando el lenguaje de programación HALIDE. Se detallará la definición del algoritmo y se presentará el diseño de cuatro esquemas de procesado diferentes. Se evaluarán el rendimiento de las cuatro implementaciones para diferentes sistemas microprocesador y se analizarán las diferencias en relación a las características técnicas de los respectivos sistemas.

1. ESTRUCTURA DEL PROGRAMA Y MODOS DE COMPILACIÓN

Tal y como se hizo en el capítulo anterior, se crea una estructura de programa similar a la que se tendrá en un caso real de aplicación, esto es el desarrollo de funciones determinadas compiladas en sus respectivos ficheros objetos y la posterior unión de todos ellos en una sola librería compartida. De este modo desarrollaremos las aplicaciones llamando a los algoritmos compilados en una librería formando una API.

Por lo tanto, a partir de un fichero Makefile se compilará el fichero fuente “denoising.cpp” que contiene la implementación del algoritmo. Finalmente se compilará el fichero “test.cpp” que hará uso del fichero objeto generado. Como ya se comentó en el capítulo dedicado a Halide, existen dos vías de compilación: compilación en tiempo de ejecución (just-in-time, JIT) y compilación estática (ahead-of-time, AOT). En este caso se utilizará la compilación estática para generar el fichero objeto con la compilación del algoritmo.

2. DEFINICIÓN DEL ALGORITMO

Los programas desarrollados con Halide se basan en dos partes: una definición del comportamiento del algoritmo, y una definición del esquema de cómputo (cómo se evalúan las funciones, cuándo se evalúan las subfunciones, dónde se guardan los valores temporales,…). Se implementa el comportamiento del algoritmo de la siguiente manera:

```
derx = input_n(x, y, c) - input_n(x-1, y, c);
dery = input_n(x, y, c) - input_n(x, y-1, c);

expr1 = Halide::fast_inverse_sqrt( Halide::fast_pow(derx, 2) +
    Halide::fast_pow(dery, 2) + 0.01f);

F1(x, y, c) = { derx * expr1, dery * expr1 };
```

```
expr2 = F1(x+1, y, c)[0] - F1(x, y, c)[0] + F1(x, y+1, c)[1]
    - F1(x, y, c)[1];

expr3 = expr2 + 2.5f*(input0(x, y, c) - input_n(x, y, c));

Z(x, y, c) = ( input_n(x, y, c) + 0.01f*expr3 )
```

Es muy importante diferenciar entre lo que es realmente un tipo Halide::`Expr` de lo que es un tipo `Halide::Func`. En el código mostrado arriba se distinguen por la sintaxis, ya que las funciones se asignan del modo `F(x, y, c)`. Por norma general un cálculo (productor) será función si sus píxeles no tienen una relación uno a uno con un único cálculo (consumidor). En nuestra implementación observamos que el único cálculo cuyos pixeles no tienen una correspondencia uno a uno con el siguiente cálculo es la función `F1`.

El último paso de esta etapa consiste en ejecutar la compilación y verificar que el resultado de la implementación realizada es el mismo que el obtenido en el modelo de referencia (en la simulación). Una vez definido el algoritmo y validado el resultado se tendrá asegurado el comportamiento independientemente de los cambios que se realicen en el esquema de procesamiento, es decir, podremos obtener peores resultados en el rendimiento de la implementación según programemos las etapas posteriores pero nunca cambiará el funcionamiento del algoritmo.

3. Definición del esquema

Se realizarán diferentes implementaciones de esquemas de programa, partiendo de los más sencillos que ejemplifican los casos extremos de localidad y redundancia. Se implementarán dos esquemas en los que no modificaremos en orden de evaluación de la función consumidora (la función Z) para analizar el comportamiento de la redundancia. Tras el estudio de estos esquemas llegaremos a las conclusiones acertadas acerca de cuándo evaluar las sub-funciones y dónde almacenar estos valores temporales (se realiza un estudio de la localidad y la redundancia). Finalmente se implementarán dos esquemas diferentes en los que se modificará el orden de evaluación de la función consumidora utilizando paralelizado y vectorización.

3.1 Evaluación en línea

Este es el esquema por defecto cuando no se indique nada. Como ya se comentó en el apartado correspondiente, consiste en substituir las llamadas a las sub-funciones por la expresión correspondiente.

```
Z.compile_to_file("denoising", {input_n0, input0});
```

Se muestra el código C (sin respetar la sintaxis) que representa la idea de este esquema, por lo que no se muestra la expresión resultante porque carece de utilidad y se ha omitido el bucle sobre la variable c (canal).

```
for( y = 0; y < 5; y++ )
      for( x = 0; x < 5; x++ )
            Z[y][x] = [EXPRESION];
```

No se ha modificado el orden en el que se evalúan los píxeles de la función consumidora (`Z`) por lo que estos serán evaluados por filas. En este caso tenemos el mejor nivel de *localidad*, pero un nivel nulo de *paralelización* y un grado muy alto de *redundancia*.

3.2 EVALUACIÓN EN LA RAÍZ

En el lado opuesto a la implementación anterior tenemos la evaluación de todas las subfunciones de manera independiente de acuerdo al orden de llamada de cada una de ellas. La implementación del esquema en Halide es la mostrada a continuación.

```
F1.compute_root();
Z.compile_to_file("denoising", {input_n0, input0});
```

Se obtiene el mejor nivel de *redundancia* (nulo) pero el peor grado de *localidad*. El código C que representa la idea de este esquema es el siguiente:

```
F1[y][x]; // Búferes de memoria

for( y = 0; y < 5; y++ )
      for( x = 0; x < 5; x++ )
            F1[y][x] = [EXPRESION];
for( y = 0; y < 5; y++ )
      for( x = 0; x < 5; x++ )
            Z[y][x] = [EXPRESION];
```

Observamos que la función `F1` se ha evaluado en su totalidad previamente a ser utilizados sus píxeles en la función consumidora `Z`. En esta implementación el nivel de *paralelizado* es nulo.

3.3 EVALUACIÓN EN MOSAICO CON CAMBIOS EN EL ORDEN DE EVALUACIÓN DE LAS SUB-FUNCIONES

Se realiza una implementación evaluando la función consumidora por *baldosas* (*tiles*), paralelización, vectorización y modificación del orden de evaluación y lugar de almacenamiento de los datos temporales de la función productora. La implementación en Halide es la siguiente.

```
Z.tile(x, y, x_outer, y_outer, x_inner, y_inner, 32, 32);
Z.fuse(x_outer, y_outer, tile_index);
Z.parallel(tile_index);
Z.vectorize(x_inner, 8);
F1.store_at(Z, tile_index).compute_at(Z, y_inner).vectorize(x, 8);
```

A través del siguiente esquema en C podemos observar claramente el efecto de modificar el lugar de cálculo de `F1` y el lugar de almacenamiento de sus valores. En este caso se evaluará toda una línea (en el segmento de la baldosa) cada vez que se empiece a evaluar el primer píxel de una línea de la *baldosa* en la función consumidora. Al situar el búfer de `F1` en la variable externa conseguimos que se almacenen las líneas necesarias para el cálculo (reducción de la redundancia), ya que para cada píxel (x, y) que se evalúe en la función consumidora `Z` se necesitarán 3 píxeles [(x, y),$(x + 1, y)$ y $(x, y + 1)$] de Z1.

```
for( tile_index = 0; tile_index < X; tile_index ++ ) // Paralelizado
      x_outer = ... ;
      y_outer = ... ;

      F1_buffer[y][x]; // Búfer circular de F1

      for( y_inner = 0; y_inner < 32; y_inner++ )

            // Calculo de F1
            for( py = y_outer; py < y_outer+2; py++ )
                  if ( y_outer > 0 && py == y_outer )
                        continue;
```

```
                    for( px = 0; px < 4; px++ )
                          F1_buffer [py&1][px] = [EXPRESION];

              for( x = 0; x < N; x++ )
                    y = y_outer * 32 + y_inner;
                    result[y][x] = [EXPRESION];
```

A pesar de que hemos eliminado mucha redundancia evaluando la función productora por filas (eliminación de redundancia en el eje x) y almacenando sus valores fuera del bucle de la variable externa (eliminación de redundancia en el eje y), no se consigue un valor nulo de redundancia debido a los solapes entre baldosas en la función productora.

3.4 EVALUACIÓN POR FILAS CON CAMBIOS EN EL ORDEN DE EVALUACIÓN DE LAS SUB-FUNCIONES

Se realiza una implementación evaluando la función consumidora por filas haciendo uso de paralelización, vectorización y modificación del orden de evaluación y lugar de almacenamiento de los datos temporales de la función productora. La implementación en Halide es la siguiente.

```
Z.split(y, y_outer, y_inner, 16);
Z.parallel(y_outer);
Z.vectorize(x, 8);
F1.store_at(Z, y_outer).compute_at(Z, y_inner).vectorize(x, 8);
```

El siguiente esquema en C muestra el comportamiento básico del esquema de evaluación por línea con modificaciones en el lugar de evaluación y el lugar de almacenamiento de los dato s temporales de la sub-función. Se crea un hilo de ejecución cada 16 filas, donde cada hilo evaluará los puntos de la función consumidora Z por filas (es decir, iterando rápidamente sobre la variable x y mas lentamente sobre la variable y). En el primer píxel que se evalúe al inicio de cada línea de la función consumidor Z se evaluarán los nuevos puntos de la función productora Z necesarios para la evaluación de esa línea.

```
for( y_outter = 0; y < M/16; y++ )  // Bucle paralelizado

      F1_buffer[2][5]; // Búfer circular de F1

      for( y_inner = 0; y < 16; y++ )
            y = y_outer * 16 + y_inner;

            for( py = y; py < y+2; py++ )
                  if ( y > 0 && py == y )
                        continue;
                  for( px = 0; px < N; px++ )
                        F1_buffer [py&1][px] = [EXPRESION];

            for( x = 0; x < N; x++ )
                  Z[y][x] = [EXPRESION];
```

Del mismo modo como sucedió en el esquema anterior, a pesar de que hemos eliminado mucha redundancia evaluando la función productora por filas (eliminación de redundancia en el eje x) y almacenando sus valores fuera del bucle de la variable externa (eliminación de redundancia en el eje y), no se consigue un valor nulo de redundancia debido a los solapes, en la función productora, de las regiones en las que se ha dividido la función consumidora. Es decir, la última línea de una región $Z(x, y+15)$ operará con los valores de las líneas $F1(x, y+1)$ y $F1(x, y+15)$ de la función productora, y la primera línea de la siguiente región $Z(x, y+16)$ necesitará de lo valores de las líneas $F1(x, y+16)$ y $F1(x, y+17)$. Al situarse el cálculo de la función productora dentro del bucle de la variable `y_outter` y haber paralelizado su

ejecución, cada hilo deberá calcular de manera independiente esa misma línea produciendo cierto nivel de redundancia. En este caso la redundancia debida a los solapes entre regiones es similar a la que se produce para el caso anterior de esquema de evaluación por mosaico.

4. COMPILACIÓN CRUZADA

El lenguaje Halide permite la compilación cruzada para otras plataformas haciendo uso de su proceso de compilación estática. Este proceso es muy sencillo en este caso, requiriéndose únicamente la definición de los siguientes parámetros:

```
Halide::Target target;
target.os = Halide::Target::Android;
target.arch = Halide::Target::ARM;
target.bits = 32;
std::vector<Halide::Target::Feature> arm_features;
target.set_features(arm_features);

Z.compile_to_file("denoising_android", {input_n0, input0}, target);
```

Se crea un objeto de clase `Halide::Target`, la cual es una estructura en la que se definirán los parámetros del sistema para el que se desea compilar (sistema operativo, arquitectura del procesador, características, etc.). Finalmente compilamos utilizando la llamada habitual, pero en este caso pasado el objeto que contiene la estructura con la definición de los parámetros de compilación (`target`). Anteriormente no se indicaba nada en este campo ya que el valor por defecto es la compilación para la propia máquina. Finalmente, se compila el fichero fuente con la definición del algoritmo y su esquema de procesado utilizando las herramientas habituales (*gcc*) de compilación en C++. Como se comentó en el capítulo dedicado al lenguaje Halide, el fichero objeto compilado contiene el propio proceso de compilación de Halide, por lo que cuando ejecutemos el fichero objeto generado se creará un nuevo fichero objeto con el resultado de la compilación para plataformas Android/ARM del algoritmo definido en Halide.

5. RESULTADOS

Se muestran a continuación los resultados obtenido en la evaluación del coste de cada implementación en término de tiempo de ejecución.

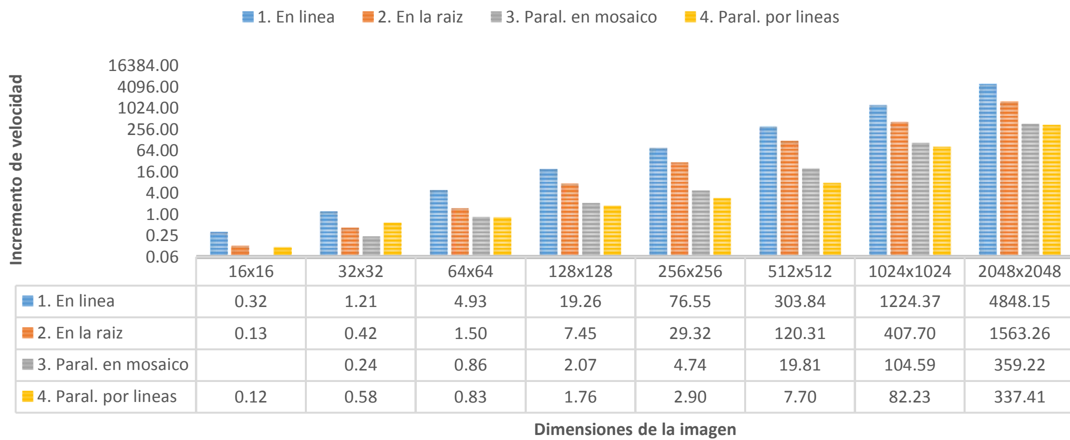
Para caracterizar el comportamiento de las diferentes implementaciones se realizan pruebas con diferentes dimensiones de imagen, para obtener de este modo la respuesta del tiempo de ejecución de cada implementación en términos del tamaño de la imagen de entrada. Nuevamente se observa que esta respuesta es, aproximadamente, dependiente de manera lineal a la dimensionalidad de la imagen de entrada.

Se observa el diferente comportamiento de las implementaciones para cada arquitectura. Estas diferencias de funcionamiento según el tipo de implementación se analizarán en el apartado siguiente, llegando a las conclusiones sobre la relación entre el esquema de procesado de cada implementación con las características físicas del sistema procesador sobre el que se trabaja.

Los sistemas microprocesadores sobre los que se evalúan las implementaciones son los correspondientes a los dos teléfonos inteligentes (Samsung Galaxy SII y Galaxy S4) descritos en el capítulo de introducción a la memoria del trabajo ya que estos dos dispositivos presentan unas características técnicas muy diferentes. De este modo nos resultará más fácil visualizar el efecto de las prestaciones técnicas sobre el esquema de procesado comparando los dos sistemas microprocesadores embebidos.

5.1 SISTEMAS OPERATIVOS GNU/LINUX SOBRE ARQUITECTURA X86-64

Se analiza el comportamiento en un sistema microprocesador de propósito general con sistema operativo GNU/Linux sobre procesador **Intel i7** con arquitectura x86-64. Como se comentó, el sistema dispone de 16GB de memoria volátil primaria. Se obtienen los resultados mostrados a en la figura 9.1.



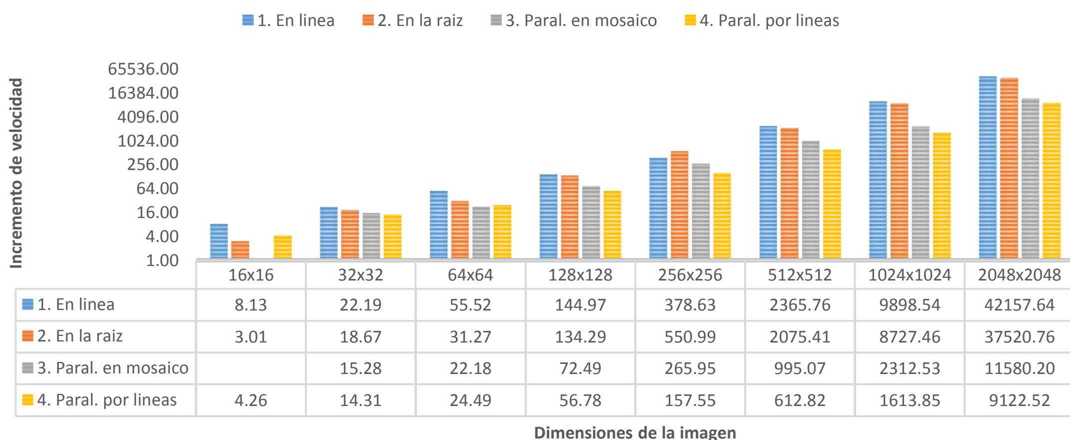
	16x16	32x32	64x64	128x128	256x256	512x512	1024x1024	2048x2048
1. En linea	0.32	1.21	4.93	19.26	76.55	303.84	1224.37	4848.15
2. En la raiz	0.13	0.42	1.50	7.45	29.32	120.31	407.70	1563.26
3. Paral. en mosaico		0.24	0.86	2.07	4.74	19.81	104.59	359.22
4. Paral. por lineas	0.12	0.58	0.83	1.76	2.90	7.70	82.23	337.41

Figura 9.1: Resultados de rendimiento obtenidos para cada esquema de procesado en arquitecturas x86-64

5.2 SISTEMAS OPERATIVOS ANDROID/LINUX SOBRE ARQUITECTURA ARMV7

En este apartado se analiza el comportamiento de las implementaciones en sistemas microprocesadores embebidos con sistema operativo Android/Linux sobre microprocesadores con arquitectura ARMv7. Resulta común en sistemas embebidos encontrar sistemas con diferencias importantes de prestaciones (tamaño y velocidad de acceso a memoria primaria, número de núcleos, etc.). En nuestro caso, para mostrar el efecto de las diferencias de prestaciones técnicas en el comportamiento de los diferentes esquemas, se evalúa el rendimiento de las implementaciones en un teléfono inteligente Samsung Galaxy SII y un Samsung Galaxy S4. El efecto de estas diferencias en las características técnicas de los sistemas se detallará en el apartado siguiente.

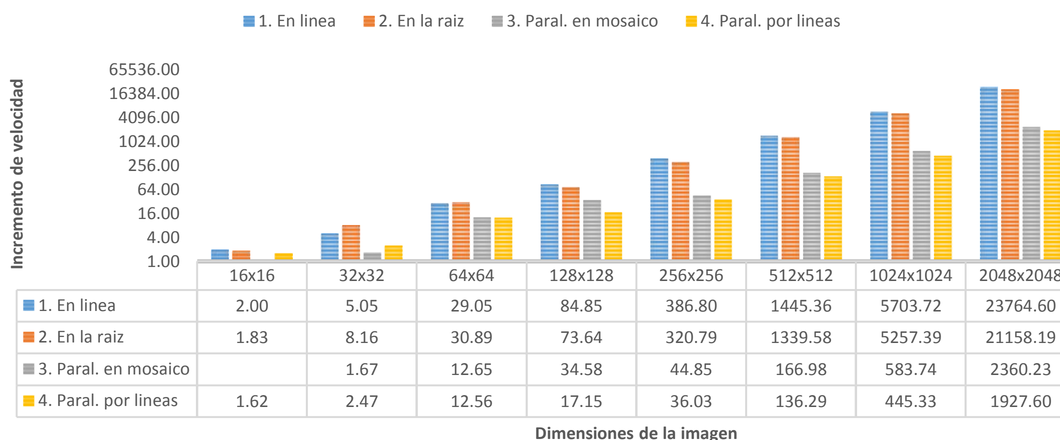
Los resultados para el **Samsung Galaxy SII** son los mostrados a en la figura 9.2.



	16x16	32x32	64x64	128x128	256x256	512x512	1024x1024	2048x2048
1. En linea	8.13	22.19	55.52	144.97	378.63	2365.76	9898.54	42157.64
2. En la raiz	3.01	18.67	31.27	134.29	550.99	2075.41	8727.46	37520.76
3. Paral. en mosaico		15.28	22.18	72.49	265.95	995.07	2312.53	11580.20
4. Paral. por lineas	4.26	14.31	24.49	56.78	157.55	612.82	1613.85	9122.52

Figura 9.2: Resultados de rendimiento obtenidos en el dispositivo Samsung Galaxy SII

Durante la evaluación de los esquemas de procesado en el teléfono inteligente **Samsung Galaxy S4** se han obtenido los resultados que se muestran en la figura 9.3.



	16x16	32x32	64x64	128x128	256x256	512x512	1024x1024	2048x2048
1. En linea	2.00	5.05	29.05	84.85	386.80	1445.36	5703.72	23764.60
2. En la raiz	1.83	8.16	30.89	73.64	320.79	1339.58	5257.39	21158.19
3. Paral. en mosaico		1.67	12.65	34.58	44.85	166.98	583.74	2360.23
4. Paral. por lineas	1.62	2.47	12.56	17.15	36.03	136.29	445.33	1927.60

Figura 9.3: Resultados de rendimiento obtenidos en el dispositivo Samsung Galaxy S4

6. VALORACIÓN DE LOS DIFERENTES ESQUEMAS

De manera global se observa en los resultados que el rendimiento, como cabía esperar, es mucho mayor en el sistema microprocesador de carácter general (ordenador portátil) ya que las prestaciones de este sistema son mucho mayores a las disponibles en los sistemas microprocesadores embebidos. Para un tamaño de imagen de 2048x2048 píxeles se obtiene una diferencia aproximada de 5 veces entre la ejecución en el PC y el teléfono inteligente.

No obstante este comportamiento global era de esperar y poco se puede hacer al respecto en términos de programación ya que el límite se encuentra en las propias características de los sistemas en su conjunto. Sin embargo, de manera particular se puede analizar el diferente comportamiento según los esquemas de programación en relación a las características de los sistemas sobre los que se desea compilar y elegir el más adecuado para la aplicación en particular.

6.1 COMPORTAMIENTO DE LA LOCALIDAD Y LA REDUNDANCIA

Los esquemas de procesados 1 y 2 (evaluación en línea y en la raíz, respectivamente) se han desarrollado con la intención de analizar los efectos de la relación entre la capacidad de procesado de los núcleos del procesador y las características de la memoria volátil (tamaño de la memoria caché, tamaño y velocidad de acceso a memoria primaria, etc.).

Se aprecia en los resultados de la evaluación en el PC que es esquema de evaluación en la raíz (*caso 2*) presenta una significativa mejoría de rendimiento en comparación con el cálculo en línea (*caso 1*). Como se ha comentado en muchos ocasiones a lo largo de la memoria, la evaluación de las sub-funciones en la raíz provoca una sobrecarga de la memoria volátil desplazando los búferes a niveles inferiores de memoria (a la memoria primaria) si se excede el tamaño de la memoria caché, mientras que el cálculo en línea produce una sobrecarga del núcleo. En el caso del PC se dispone de una memoria caché relativamente grande (L2 de 6MB) por lo que seguramente gran parte (según el tamaño de la imagen) de los búferes temporales se encuentren en ella en el esquema de evaluación en la raíz, teniendo además de un acceso rápido a memoria primaria. Es por este motivo que la evaluación en la raíz presenta mejores prestaciones que la evaluación en línea.

Sin embargo esto no sucede en los sistemas microprocesadores embebidos analizados, los cuales tienen unos tamaños de caché reducidos (L2 de 512KB en el caso del Samsung Galaxy SII y de 2MB en el caso del Galaxy S4) y unos accesos a memoria primaria lentos. Es por

este motivo que la mejoría experimentada en el sistema PC no se experimenta en los sistemas embebidos analizados. Puede observarse en los resultados como el esquema del caso 2 (en la raíz) presenta unos resultados muy similares a los obtenidos en el esquema del caso 1 (en línea).

Observamos también diferencias de rendimiento en los casos 2 y 1 para los dos sistemas embebidos. En los resultados obtenidos para el Samsung Galaxy SII se observa una ligera mejora del rendimiento en el caso 2 (en línea) respecto al caso 1 (en la raíz), y sin embargo esta mejora del rendimiento es mucho menor en los resultados del Galaxy S4. Este efecto es producido por la capacidad de cómputo de los respectivos procesadores. El Galaxy SII tiene 2 núcleos a una frecuencia de reloj de 1500MHz mientras que el Galaxy S4 tiene 4 núcleos y una frecuencia de reloj de 1900MHz. Esta importante mejora en las características técnicas del núcleo de procesador hace que el caso 1 se (en línea) se comporte mucho mejor en el caso del Galaxy S4, teniendo un comportamiento muy similar al caso 2 (en la raíz).

6.2 COMPORTAMIENTO DE LA PARALELIZACIÓN

Se observa, de manera general, que los casos 2 y 3 presentan un nivel de mejora del rendimiento imporante. Realizaremos una comparación entre el caso 1 (evaluación en línea) y el caso 3 (evaluación en mosaico) para observar el efecto de la paralelización, aunque hay que considerar que en el caso 3 la evaluación de las sub-funciones no se realiza en línea por lo que parte de esta mejoría se debe al uso eficiente de la memoria caché. Se presenta en la figura 9.4 el incremento del rendimiento entre el caso 1 y el caso 3 sobre el tiempo de procesado para una imagen con un tamaño de 2048x2048 píxeles.

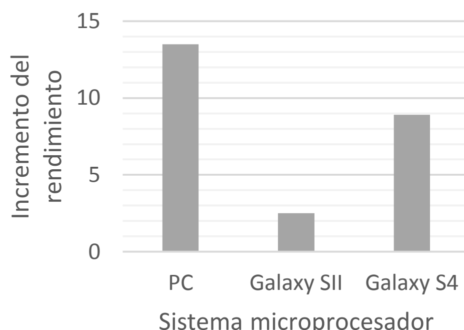


Figura 9.4: Impacto de las técnicas de paralelizado

Tras la incorporación de técnicas de paralelización y vectorización en los esquemas se saca el máximo partido a los microprocesadores multi-núcleo. En el caso del Samsung Galaxy SII se dispone únicamente de dos núcleos (*dual-core)* por lo que se observa una discreta mejora del rendimiento. No obstante en el caso del Samsung Galaxy S4 se dispone de cuatro núcleos (*quad-core*) por lo que la mejor en este caso es mucho mayor al disponerse de unas características técnicas de microprocesador que favorecen el uso de la paralización de código. Finalmente se observa una mejora mucho mayor en el caso del microprocesador Intel i7, el cual dispone de 4 núcleos y 8 hilos de ejecución a una velocidad de reloj de 2300MHz.

Capítulo 10. Conclusiones y líneas futuras

1. CONCLUSIONES

Se ha podido comprobar que la implementación de un algoritmo de procesado de imagen utilizando las técnicas habituales es un proceso costoso. Si bien se disponen de diversas de librerías que cuentan con una API amigable y de fácil codificación, no todas ellas están enfocadas al campo de la visión artificial. La gran mayoría de librerías están desarrolladas de una manera generalista y abarcan un campo amplio del cálculo lineal. Estas librerías (*Eigen* y *Armadillo* en nuestro caso práctico) nos han permitido realizar una optimización sobre las partes de código referidas a operaciones sobre matrices, sin embargo en el uso de funciones más específicas de procesado de imagen (como ha sido el caso del filtro en dos dimensiones) se ha tenido que recurrir a un conjunto muy reducido de librerías de visión artificial (en nuestro caso *OpenCV*). Esto nos provoca que, en mayor o menor grado, dependamos de un conjunto limitado de librerías por lo que las alternativas de implementación se reducen. También se añade la dificultad de tener que trabajar con varias librerías en una misma implementación, por lo que hay que ser cuidadoso en el manejo de la interoperabilidad de las diferentes librerías. También se ha demostrado que cabe la posibilidad de codificar manualmente ciertas funciones de una manera optimizada haciendo uso de técnicas avanzadas de programación en C++ o haciendo uso de librerías generalistas de optimización (bien aplicando paralelización con liberías específicas como TBB y OpenMP, bien aplicando parametrización de código y optimización de los parámetros mediante OpenTuner, etc.).

Sin embargo, el lenguaje de programación específico Halide nos ha permitido desarrollar el algoritmo invirtiendo un tiempo de desarrollo mucho menor. Este lenguaje está construido específicamente para el campo de la visión artificial por lo que su manejo es muy cómo e intuitivo. La separación entre los conceptos de algoritmo y esquema de procesado es clave en este nuevo lenguaje. Esta separación nos ha permitido definir el comportamiento de nuestro algoritmo y asegurarnos del correcto resultado, y a partir de aquí realizar todas las pruebas necesarias sobre el esquema de procesado en busca de la implementación óptima para cada sistema con la seguridad de que nunca vamos a modificar el resultado del algoritmo definido.

Otro aspecto muy importante es la facilidad de compilación cruzada que ofrece Halide con soporte para las plataformas más comunes. El paso de compilación cruzada para sistemas ARM nos ha resultado un proceso complicado y laborioso en el caso de utilización de librerías, habiendo un caso en el que se ha tenido que desistir del intento debido al grado de complejidad (en concreto en las librerías *Armadillo* + *OpenBLAS*). De este modo, y junto a la facilidad de desarrollo de diferentes esquemas de procesado, encontramos que Halide nos va a permitir adaptar un mismo algoritmo a diferentes sistemas microprocesador de una manera cómoda y sencilla.

Finalmente, y quizás el aspecto fundamental, observamos un rendimiento excelente en las implementaciones realizadas con Halide. Se aprecia un incremento del rendimiento en términos de tiempo de ejecución de hasta 12 veces en comparación con el mejor caso obtenido mediante el uso de librerías.

2. FUTURAS LÍNEAS DE INVESTIGACIÓN

Los resultados obtenidos despiertan un gran interés en el lenguaje Halide, por lo que sería recomendable ampliar el conocimiento del lenguaje en profundidad ya que se disponen de multitud de funciones que permiten incrementar el grado de optimización. Se plantea continuar con la implementación de algoritmos más complejos que nos permitan buscar posibles aspectos negativos.

A. Referencias

1. Referencias

1 Armadillo Library. (Septiembre de 2015). *Sitio web oficial*. Obtenido de http://arma.sourceforge.net/

2 ATLAS Library. (Septiembre de 2015). *Sitio web oficial*. Obtenido de http://math-atlas.sourceforge.net/

3 Carnegie Mellon University. (Septiembre de 2015). *Human Sensing Lab*. Obtenido de http://www.humansensing.cs.cmu.edu/

4 CMake. (Septiembre de 2015). *Sitio web oficial*. Obtenido de http://www.cmake.org/

5 Eigen Library. (Septiembre de 2015). *Sitio web oficial*. Obtenido de http://eigen.tuxfamily.org/

6 GNU Project. (Septiembre de 2015). *About GNU Make*. Obtenido de https://www.gnu.org/software/make/

7 HALIDE. (Septiembre de 2015). *Lecture about HALIDE*. Obtenido de http://halide-lang.org/assets/lectures/Halide1.pdf

8 HALIDE. (Septiembre de 2015). *Sitio web oficial*. Obtenido de http://halide-lang.org/

9 Ho, N. (Septiembre de 2015). *Pruebas de rendimiento de diferentes librerías*. Obtenido de http://nghiaho.com/?p=1726

10 KDE Forum. (Septiembre de 2015). *Convolution between two matrices (images) with Eigen*. Obtenido de https://forum.kde.org/viewtopic.php?f=74&t=96407

11 Linux Kernel Organization, Inc. (Septiembre de 2015). *The Linux Kernel Archives*. Obtenido de http://www.kernel.org/

12 NVIDIA. (Septiembre de 2015). *Especificaciones Jetson TK1*. Obtenido de https://developer.nvidia.com/jetson-tk1

13 OpenBLAS Library. (Septiembre de 2015). *Sitio web oficial*. Obtenido de http://www.openblas.net/

14 OpenCV. (Septiembre de 2015). *Sitio web oficial*. Obtenido de http://www.opencv.org/

15 Rudin, L., Osher, S., & Fatemi, E. (1992). Nonlinear total variation based noise removal algorithms. *Journal Physica D Volume 60 Issue 1-4*, 259-268.

16 Yaghmour, K. (2013). *Embedded Android.* O'Reilly.

Anexo 1. Implementación en Matlab

```
function J=tv(I,iter,dt,ep,lam,I0)
%% Total Variation denoising.
%% Example: J=tv(I,iter,dt,ep,lam,I0)
%% Input: I    - image (double array gray level 1-256),
%%        iter - num of iterations,
%%        dt   - time step [0.2],
%%        ep   - epsilon (of gradient regularization) [1],
%%        lam  - fidelity term lambda [0],
%%        I0   - input (noisy) image [I0=I]
%%       (default values are in [])
%% Output: evolved image

[ny,nx]=size(I); ep2=0.05;%ep^2;

for i=1:iter,  %% do iterations
    % estimate derivatives
    I_x = I(:,[2:nx nx]) - I;
       I_y = I([2:ny ny],:) - I;
    I_xI = I - I(:,[1 1:nx-1]);
       I_yI = I - I([1 1:ny-1],:);
       %I_x = (I(:,[2:nx nx])-I(:,[1 1:nx-1]))/2;
       %I_y = (I([2:ny ny],:)-I([1 1:ny-1],:))/2;

    minmod_x = ((sign(I_x)+sign(I_xI))/2).*min(I_x,I_xI);
    minmod_y = ((sign(I_y)+sign(I_yI))/2).*min(I_y,I_yI);

    U = I_x ./ sqrt( ep2 + I_x.^2 + minmod_y.^2 );
    V = I_y ./ sqrt( ep2 + minmod_x.^2 + I_y.^2 );

    %U_x = U(:,[2:nx nx]) - U;
    %V_y = V([2:ny ny],:) - V;
    U_x = U - U(:,[1 1:nx-1]);
    V_y = V - V([1 1:ny-1],:);
    %U_x = (U(:,[2:nx nx]) - U(:,[1 1:nx-1]))/2;
    %V_y = (V([2:ny ny],:) - V([1 1:ny-1],:))/2;

   I_t = U_x + V_y + lam.*(I0-I);
   I=I+dt*I_t;  %% evolve image by dt
end % for i
%% return image
J=I;
```

Anexo 2. Implementación con OpenCV

```
#include <opencv2/photo/photo.hpp>
#include <opencv2/core/core.hpp>

using namespace cv;

Mat denoise_opencv(Mat src, float alpha) {

	Mat input_n;
	input_n = src/255;

	// Parametros algoritmo
	//float alpha = 2.5;
	float dt = 0.01;
	float sigma = 0.01f;

	// Parametros convolucion
	Mat kernelx = (Mat_<float>(1,2)<<-1, 1);
	Mat kernely = (Mat_<float>(2,1)<<-1, 1);
	Point anchorx_r = Point( -1, 0 );
	Point anchory_r = Point( 0, -1 );
	Point anchorx_l = Point( 0, 0 );
	Point anchory_l = Point( 0, 0 );
	double delta = 0;
	int ddepth = -1;

	// Inicializacion matrices
	Mat fx(src.rows,src.cols,CV_32F);
	Mat fy(src.rows,src.cols,CV_32F);
	Mat Module(src.rows,src.cols,CV_32F);
	Mat U1(src.rows,src.cols,CV_32F);
	Mat U2(src.rows,src.cols,CV_32F);
	Mat V1(src.rows,src.cols,CV_32F);
	Mat V2(src.rows,src.cols,CV_32F);
	Mat W(src.rows,src.cols,CV_32F);

	for (int i = 0; i < 25; i++) {

		filter2D(input_n, fx, ddepth, kernelx, anchorx_r, delta, BORDER_REPLICATE );
		filter2D(input_n, fy, ddepth, kernely, anchory_r, delta, BORDER_REPLICATE );

		Module = fx.mul(fx) + fy.mul(fy) + sigma;
		sqrt(Module, Module);

		U1 = fx / Module;
		U2 = fy / Module;

		filter2D(U1, V1, ddepth, kernelx, anchorx_l, delta, BORDER_REPLICATE );
		filter2D(U2, V2, ddepth, kernely, anchory_l, delta, BORDER_REPLICATE );

		W = V1 + V2 + src.mul(alpha/255) - input_n.mul(alpha);

		input_n = input_n + W.mul(dt);
	}

	return input_n.mul(255);

}
```

Anexo 3. Implementación con Eigen

```
#include <opencv2/photo/photo.hpp>
#include <opencv2/core/core.hpp>
#include <eigen3/Eigen/Dense>
#include <eigen3/Eigen/Core>

using namespace cv;

Mat denoise_eigen(Mat src, float alpha) {

	Mat input_n;
	input_n = src/255;

	// Parametros algoritmo
	//float alpha = 2.5;
	float dt = 0.01;
	float sigma = 0.01f;

	// Parametros convolucion
	Mat kernelx = (Mat_<float>(1,2)<<-1, 1);
	Mat kernely = (Mat_<float>(2,1)<<-1, 1);
	Point anchorx_r = Point( -1, 0 );
	Point anchory_r = Point( 0, -1 );
	Point anchorx_l = Point( 0, 0 );
	Point anchory_l = Point( 0, 0 );
	double delta = 0;
	int ddepth = -1;

	// Inicializacion matrices
	Mat fx(src.rows,src.cols,CV_32F);
	Mat fy(src.rows,src.cols,CV_32F);
	Mat Module(src.rows,src.cols,CV_32F);
	Mat U1(src.rows,src.cols,CV_32F);
	Mat U2(src.rows,src.cols,CV_32F);
	Mat V1(src.rows,src.cols,CV_32F);
	Mat V2(src.rows,src.cols,CV_32F);
	Mat W(src.rows,src.cols,CV_32F);

	// Eigen map
	Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
> Module_mappedMat ((float *)Module.data, src.rows, src.cols);
	Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
> fx_mappedMat ((float *)fx.data, src.rows, src.cols);
	Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
> fy_mappedMat ((float *)fy.data, src.rows, src.cols);
	Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
> U1_mappedMat ((float *)U1.data, src.rows, src.cols);
	Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
> U2_mappedMat ((float *)U2.data, src.rows, src.cols);
	Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
> V1_mappedMat ((float *)V1.data, src.rows, src.cols);
	Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
> V2_mappedMat ((float *)V2.data, src.rows, src.cols);
	Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
> W_mappedMat ((float *)W.data, src.rows, src.cols);
	Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
> inputn_mappedMat ((float *)input_n.data, src.rows, src.cols);
	Eigen::Map<Eigen::Matrix<float, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
> src_mappedMat ((float *)src.data, src.rows, src.cols);

	for (int i = 0; i < 25; i++) {
```

```
            filter2D(input_n, fx, ddepth, kernelx, anchorx_r, delta, BORDER_REPLICATE
);
            filter2D(input_n, fy, ddepth, kernely, anchory_r, delta, BORDER_REPLICATE
);

            Module_mappedMat    =    ((fx_mappedMat.cwiseProduct(fx_mappedMat)    +
fy_mappedMat.cwiseProduct(fy_mappedMat)).array()+sigma).cwiseSqrt();

            U1_mappedMat = fx_mappedMat.cwiseQuotient(Module_mappedMat);
            U2_mappedMat = fy_mappedMat.cwiseQuotient(Module_mappedMat);

            filter2D(U1, V1, ddepth, kernelx, anchorx_l, delta, BORDER_REPLICATE );
            filter2D(U2, V2, ddepth, kernely, anchory_l, delta, BORDER_REPLICATE );

            W_mappedMat = V1_mappedMat + V2_mappedMat + src_mappedMat*(alpha/255.0f)
- inputn_mappedMat*alpha;

            input_n = input_n + W.mul(dt);
      }

      return input_n.mul(255);

}
```

Anexo 4. Implementación con Armadillo

```
#include <opencv2/photo/photo.hpp>
#include <opencv2/core/core.hpp>
#include <armadillo>

using namespace cv;

Mat denoise_arma(Mat src, float alpha) {

	Mat input_n;
	input_n = src/255;

	// Parametros algoritmo
	//float alpha = 2.5;
	float dt = 0.01;
	float sigma = 0.01f;

	// Parametros convolucion
	Mat kernelx = (Mat_<float>(1,2)<<-1, 1);
	Mat kernely = (Mat_<float>(2,1)<<-1, 1);
	Point anchorx_r = Point( -1, 0 );
	Point anchory_r = Point( 0, -1 );
	Point anchorx_l = Point( 0, 0 );
	Point anchory_l = Point( 0, 0 );
	double delta = 0;
	int ddepth = -1;

	// Inicializacion matrices
	Mat fx(src.rows,src.cols,CV_32F);
	Mat fy(src.rows,src.cols,CV_32F);
	Mat Module(src.rows,src.cols,CV_32F);
	Mat V1(src.rows,src.cols,CV_32F);
	Mat V2(src.rows,src.cols,CV_32F);
	Mat U1(src.rows,src.cols,CV_32F);
	Mat U2(src.rows,src.cols,CV_32F);
	Mat W(src.rows,src.cols,CV_32F);

	// Armadillo matrix
	arma::fmat arma_W( W.ptr<float>(), W.rows, W.cols, false, true );
	arma::fmat arma_src( src.ptr<float>(), src.rows, src.cols, false, true );
	arma::fmat  arma_inputn(  input_n.ptr<float>(),  input_n.rows,  input_n.cols,  false,
true );
	arma::fmat arma_fx( fx.ptr<float>(), fx.rows, fx.cols, false, true );
	arma::fmat arma_fy( fy.ptr<float>(), fy.rows, fy.cols, false, true );
	arma::fmat arma_Module( fy.ptr<float>(), Module.rows, Module.cols, false, true );
	arma::fmat arma_U1( U1.ptr<float>(), U1.rows, U1.cols, false, true );
	arma::fmat arma_U2( U2.ptr<float>(), U2.rows, U2.cols, false, true );
	arma::fmat arma_V1( V1.ptr<float>(), V1.rows, V1.cols, false, true );
	arma::fmat arma_V2( V2.ptr<float>(), V2.rows, V2.cols, false, true );

	for (int i = 0; i < 25; i++) {

		filter2D(input_n, fx, ddepth, kernelx, anchorx_r, delta, BORDER_REPLICATE );
		filter2D(input_n, fy, ddepth, kernely, anchory_r, delta, BORDER_REPLICATE );

		arma_Module = sqrt((arma_fx%arma_fx + arma_fy%arma_fy)+sigma);

		arma_U1 = arma_fx/arma_Module;
		arma_U2 = arma_fy/arma_Module;

		filter2D(U1, V1, ddepth, kernelx, anchorx_l, delta, BORDER_REPLICATE );
```

```
            filter2D(U2, V2, ddepth, kernely, anchory_l, delta, BORDER_REPLICATE );
            arma_W = arma_V1 + arma_V2 + arma_src*(alpha/255.0) - arma_inputn*alpha;
            arma_inputn += arma_W*dt;
        }
        return input_n.mul(255);
}
```

Anexo 5. Implementación con HALIDE

```
#include "Halide.h"

int main(int argc, char **argv) {

      Halide::ImageParam input_n0(Halide::Float(32), 3, "input_n0");
      Halide::ImageParam input0(Halide::Float(32), 3, "input0");

      // Variables
      Halide::Var  x("x"),  y("y"),  c("c"),  x_outer("x_outer"),  y_outer("y_outer"),
x_inner("x_inner"),         y_inner("y_inner"),          x_inner_outer("x_inner_outer"),
y_inner_outer("y_inner_outer"),       x_vectors("x_vectors"),       y_pairs("y_pairs"),
tile_index("tile_index"), subtile_index;

      // Funciones
      Halide::Func F1("F1"), Z("Z");
      Halide::Expr derx, dery, expr1, expr2, expr3;

      // Cmportamiento en los bordes
      Halide::Func input_n = Halide::BoundaryConditions::repeat_edge(input_n0);

      // ALGORITMO -----------------------
      derx = input_n(x, y, c) - input_n(x-1, y, c);
      dery = input_n(x, y, c) - input_n(x, y-1, c);

      expr1    =    Halide::fast_inverse_sqrt(    Halide::fast_pow(derx,    2)    +
Halide::fast_pow(dery, 2) + 0.01f);

      F1(x, y, c) = { derx * expr1, dery * expr1 };

      expr2 = F1(x+1, y, c)[0] - F1(x, y, c)[0] + F1(x, y+1, c)[1] - F1(x, y, c)[1];

      expr3 = expr2 + 2.5f*(input0(x, y, c) - input_n(x, y, c));

      Z(x, y, c) = ( input_n(x, y, c) + 0.01f*expr3 );

      // ESQUEMA -----------------------
      // 1) Codigo insertado
      // Por defecto

      // 2) Evaluacion en la raiz
      //F1.compute_root();

      // 3) Evaluacion en mosaico
      //Z.tile(x, y, x_outer, y_outer, x_inner, y_inner, 32, 32);
      //Z.fuse(x_outer, y_outer, tile_index);
      //Z.parallel(tile_index);
      //Z.vectorize(x_inner, 8);
      //F1.store_at(Z, tile_index).compute_at(Z, y_inner).vectorize(x, 8);

      // 4) Evaluacion por filas
      //Z.split(y, y_outer, y_inner, 16);
      //Z.parallel(y_outer);
      //Z.vectorize(x, 8);
      //F1.store_at(Z, y_outer).compute_at(Z, y_inner).vectorize(x, 8);

      Z.compile_to_file("denoising", {input_n0, input0});

      return 0;
}
```